The Pennsylvania State University
The Graduate School

SOFTWARE BASED TECHNIQUES FOR ROBUST COMPUTING ON CHIP MULTIPROCESSORS

A Dissertation in
Computer Science and Engineering
by
Sri Hari Krishna Narayanan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2008

The dissertation of Sri Hari Krishna Narayanan was reviewed and approved* by the following:

Mahmut Taylan Kandemir
Assistant Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Mary Jane Irwin
Evan Pugh Professor of Computer Science and Engineering

Padma Raghavan
Professor of Computer Science and Engineering

Tracy Mullen
Assistant Professor of Information Sciences and Technology

Boyana Norris
Computer Scientist, Argonne National Laboratory
Special Member

Raj Acharya
Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Robustness and performance are two requirements of any computing system. They may be quantified by different metrics depending on the domain of Computer Science being discussed. The loss of either robustness or performance can represent a cost to the user or owner of the system. While it is possible that robustness and performance are interdependent, often one must be sacrificed for the other. Chip Multiprocessors (CMPs) have emerged as one of key architectures for current systems. Hence this thesis proposes and evaluates sofware based techniques to improve the robustness of CMP based systems. The techniques used involve analyzing the applications and then modifying them according to the desired robustness metric. While several hardware based techniques exist to alleviate several robustness concerns, analyzing an application's behvior allows these methods to be called on less frequently or in some cases tunes the usage of the hardware method.

The three aspects of robustness that are studied in this thesis are: reduction of thermal emergencies, protection against soft errors and the security of data accessed by applications. In order to prevent excessive reliance on a hardware based cooling mechanism the proposed methods in this thesis map and schedule the application threads such that the power density on any processor or router in the architecture is reduced. The detection of soft errors in the memory is improved by carefully changing the temporal order of computations in the different redundant threads in a CMP. Results show that there is an increase in the number of soft errors detected by the proposed approach. The security of data accessed by mutually untrusting parallel threads is studied and an algorithm to perform workload balancing between these threads is presented. The algorithm is successful in improving reducing the standard deviation of the workload assigned to the different threads. Trade-offs between application performance and the confidentiality of data accessed by it in a scratch pad memmory based system are studied. A reuse based mechanism is used to tune the application based on the desired level of per-

formance and data confidentiality. An Integer Linear Prograramming approach is extends this study to applications that must complete by a deadline.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The focus of this thesis is to enable robust computing in the face of emerging challenges in chip multiprocessors (CMPs). Robustness is defined as the capability of to perform without failure under a wide range of conditions. A primary requirement that any user places on a computer system whether is that they work without failure or interruption. A secondary need is that these systems should work as fast as possible. One of the fundamental problems is that performance and robustness are divergent goals. On the one hand, the most robust computer system is one that is not turned on, and on the other hand, the most high-performing system is the one susceptible to most failures.

Robustness can be interpreted differently in different contexts. In the context of networks, robustness is the quality of fault tolerance provided by them [1]. That is, networks should either not show any effect of a fault or should degrade gracefully in the face of that fault. The term autonomic computing is used to describe the self managing, self organizing and self healing nature of certain systems such as high-end servers [2]. Autonomic systems are expected to show resilience (robustness) to failures in one or more of the components of the systems while meeting guaranteed levels of performance. Robustness in the context of secure systems is the ability to withstand attacks to the confidentiality and integrity of information [3]. Robustness in the context of software systems is the ability to provide meaningful output under all input, regardless of whether the input was

expected or not [4].

The requirement for rubustness arises from the financial losses that occur when computer systems fail. For example, an outage in the popular auction site eBay in 1999 led to a loss of millions of dollars in revenue for that firm. On average, the average cost of a single hour of application downtime is about $45,000 per hour [5]. This cost is rises exponentially with the importance of the application. Many computer systems are used to handle real-time critical operations such as flight control. Clearly, a failure in such systems can be catastrophic. Therefore, ensuring the robustness of a computer system is very important.

The scope of this work is to improve certain aspects of the robustness of modern computer systems. Modern architectures are relentlessly heading toward increasing levels of parallelism. There is a clear trend towards increasing number of processors on chip and the number of application threads running together in parallel. Such architectures have been classified as CMPs [6, 7, 8, 9, 10, 11]. In essence a CMP based architecture is one that has multiple processors on single chip that may communicate with each other through a communication bus or on-chip network. As CMP based architectures grow increasingly popular, solutions that are targeted at them will have wide ranging impact if adopted by designers and architects. Keeping this in mind, the focus of this work is to study robustness in the context of CMP based architectures from different perspectives.

The three threats to robustness in CMPs that are studied in this work are thermal emergencies, software faults, and the security of data and code. It will be shown how each threat can adversely impact the proper execution of an application an thus need to be addressed. Thermal emergencies can cause hardware faults, security breaches can cause loss if intellectual capital or valuable data and errors due to atmospheric particle strikes can lead to correctness issues in the results of computation. There are many solutions that have been proposed to address these threats to robustness at the hardware level which include [12, 13, 14, 15, 16]. While these solutions are definitely useful, solutions at the software level exploit knowledge about the application behavior as well. Exploiting, application knowledge can allow us to either avoid using hardware failsafes or require them to be used less often. Therefore, in this work we concentrate on software solutions.

In the context of modern architectures, a rise in performance is brought about

by increasing the number of transistors on chip. An increased number of transistors on roughly the same area of the chip leads to a rise in the power density on the chip. High power densities lead to high on chip temperatures which can lead to irrecoverable hardware damage. This problem and the proposed solutions will be studied in the context of CMP and Network-on-Chip (NoC) architectures in Chapters 2 and 3, respectively.

Soft errors are a growing threat to the correct execution of an application [17, 18, 19] and increased scaling of technology has exacerbated this problem [20]. In chip multiprocessor (CMP) architectures, redundant-threading (RT) is one of the ways to overcome soft errors [16]. In an RT framework the same code is simultaneously executed across all the processors and periodically the results are compared to check if the computed results across the different threads agree. The disadvantage is that in a CMP that is based on the shared memory concept, threads that operate simultaneously in the RT framework would read the same data from memory in close temporal proximity. Chapter 4 proposes an automatic versioning algorithm to create multiple versions of a thread. the access data in different temporal orders.

Finally, this work also deals with an important issue in any computing system, which is of security. In particular, the examined scenario deals with the security of an application's data; when the application is run in parallel, by multiple hosts whose processes run on the different processors of the CMP. In a security system that uses multi-level security, different hosts/threads have different capability levels and different portions of the data have differing levels of sensitivity. In such a system, it is important to ensure that no leak of sensitive data occurs and that the potential for parallel execution of the application is boosted as far as possible to reduce the execution time. Chapter 5 proposes a method to intelligently partition the data across the processors to hasten the application execution while ensuring that data remains secure.

An important method of keeping data in the memory secure is through encryption. That is, data when it resides in the memory is in encrypted form which prevents an attacker from reading the data. In order for the encrypted data to be used by the processor, the data is decrypted before it is brought to the processor and encrypted before it is written back from the processor to the memory.

Overheads associated with decryption and encryption can place a severe strain on the performance of the application. Relaxing the requirement that all the data in the memory should be in encrypted form can alleviate this situation. Chapter 6 discusses a proposed approach to allow application designers to tune their applications based on the trade-off between the desired level of performance and the desired level of data security. That is, we allow the application designer to explore a middle ground between complete robustness in terms or security and complete robustness in terms of performance expectations. The architecture in this proposed approach is a uni-processor system that contains a Scratch-Pad Memory (SPM).

A related problem is that of ensuring that the execution of the application to complete before a hard deadline while providing the maximum confidentiality possible. Chapter 7 presents an Integer Linear Programming (ILP) based approach to solving this problem. We also explore the dual of obtaining the minimum execution time for a specified level of confidentiality. Chapter 8 discusses the proposed future work.

Chapter 2

Temperature-Sensitive Loop Parallelization for Chip Multiprocessors

2.1 Introduction

Power dissipation has emerged as one of the critical constraints preventing hardware designers and software writers from extracting the maximum performance from a computer system. Prior research [21, 22, 13, 12] notes that one of the important consequences of high power consumption is power density, which in turn leads to temperature-related problems. In fact, extrapolating the changes in microprocessor organization and the device miniaturization, one can project future power dissipation density of 200 W/cm^2 [23]. This requires extensive efforts on cooling techniques and technologies.

Robustness in this context means that either temperature related problems should be prevented from occuring, or that when they occur methods should exist to safeguard the hardware and ensure that the execution is not adversely affected. While hardware solutions to temperature management problem are very important, software can also play an important role in this picture. This is because it is mainly the software that determines the circuit components exercised during the execution and the period of time for which they are exercised. In particular, compilers

determine the data and instruction access patterns (sequences) of applications, which shape the power density profile during the course of execution. Further, by using a software based method, the hardware mechanisms can be used as a failsafe rather than as a first line of defense. This may eliminate the need for certain expensive hardware mechanisms as well.

We present and evaluate three temperature-sensitive loop parallelization strategies for array-intensive applications executed on chip multiprocessors. The proposed strategies start with a pre-parallelized code and re-distribute loop iterations across processors – at compile time – in such a fashion that the temperature of each processor core is reduced, without affecting performance and power consumption very adversely. To do this, we first divide the iteration space of the loop nest being optimized into multiple chunks of equal sizes, and for each chunk, our approaches determine the best processor to use, taking into account load balance, temperature, and data locality. Our first approach employs a *temperature-driven scheduling* algorithm for re-distributing loop iterations across processors. At each step, this approach selects the coolest processor to execute the next iteration chunk so that the overall temperature can be reduced. This temperature-driven approach performs well in terms of thermal behavior as well as load balance (which is a side-benefit of optimizing temperature), but it could also lead to degraded data locality since it does not consider any data reuse when scheduling the iteration chunks. Such degraded data locality can incur significant performance and power consumption overheads, which are undesirable for on-chip multiprocessors. In comparison, our second approach, which employs a *locality-driven scheduling* algorithm, selects a processor at each step for the next iteration chunk in such a fashion that data locality can be optimized, subject to specific thermal constraints. However, this locality-driven approach is not as good as the temperature-driven approach in terms of thermal behavior and load balance. Considering the pros and cons of these two approaches, we propose a *combined approach*, which is the main contribution of this work, that has the advantages of both temperature-driven and locality-driven scheduling. In the combined approach, we first employ a temperature-driven approach to determine for each processor its state (i.e., running or idle) at each time slot. After that, we use a locality-driven approach to determine the iteration chunks to be executed on each processor so that data locality is optimized as much

as possible. Therefore, our combined approach is expected to perform well in terms of temperature, load-balancing, and data locality.

To test the effectiveness of our combined approach, we conducted experiments with five applications. Our results reveal that the proposed parallelization strategy is very successful in practice. Specifically, it contains the peak temperature of the processors under a threshold temperature with little performance/energy penalties. In addition, the proposed approach reduces the peak temperature (average temperature) by 20.9 degrees (4.3 degrees) when averaged over all the applications tested, incurring extra performance/power penalties within a small range. Our results also show that the temperature savings we achieve can be further improved by tuning the parameters of our algorithm.

The remainder of this chapter is organized as follows. A discussion of the related work is given in Section 2.2. The details of the proposed parallelization strategy are presented in Section 2.3. The results from our experimental analysis are discussed in Section 2.4. Our concluding remarks are given in Section 2.5.

2.2 Related Work

Chip multiprocessors are architectures with multiple processors on a single chip [6]. Prior architecture-related studies in the context of chip multiprocessors include [7, 8, 9, 10, 11]. Reference [24] identifies the ideal number of processors to use when optimizing a particular execution to minimize runtime or the energy consumption.

Temperature hotspots have been identified as an important design concern in modern processors [12, 25, 26]. Research on solving this problem has mainly focused on runtime techniques. In the generic case, this involves some sort of sampling of the chip temperature during execution and taking action to prevent damage to the chip if the temperature approaches the danger mark through techniques such as [12] which regulates the amount of execution that a processor performs. Other techniques involve hotspot reduction through dynamic power management which is in turn achieved by techniques such as voltage scaling.

Activity migration [13] reduces temperature in a chip by dynamically ping-ponging jobs on multiple processing elements. This method migrates computation to a different part of the die if the temperature of one processing unit goes past a

certain limit. [14] examines static thermal aware task allocation under real time constraints, by using synthesis to allocate jobs to different processing elements. The jobs are represented as a task graph and the underlying architecture itself is optimized for jobs to be run on it.

Our work is different from these in that it is static in its approach and it targets parallel loops running on chip multiprocessors. All decisions on task assignments to processors are made at compile time. While we do allocate jobs to processors at specific times and keep them idle at others, this allocation is not done as a reaction to a runtime event. Furthermore, we optimize the cache data reuse in assigning jobs to processors by calculating the overlap of existing data in cache and the data that will be accessed by the task to be allocated to that processor. This is possible because the decision to migrate computation is not taken in the face of a thermal emergency, but rather in a calculated proactive manner dictated by the compiler.

2.3 Our Approach

In this section, we first discuss the preliminaries, including the relationship between loop parallelization and the schedule table, processor count limit, and temperature constraint. Then, we present two loop scheduling algorithms, namely, *temperature-driven loop scheduling* and *locality-driven loop scheduling*. After that, we discuss our loop scheduling algorithm, which is the main contribution of this work, that combines these two algorithms and optimizes temperature, locality, and load balance.

2.3.1 Preliminaries

The architecture being simulated is an 8 core chip multiprocessor. The clock frequency of each core is 300MHz. The processors are arranged as two rows of four processors each. The L1 caches are on chip and private to each processor with a size of 16KB. The L2 cache is shared between all processors and is 2MB in size. The details of this architecture are summarized in Table 2.1.

The loop parallelization problem can be regarded as a process of *partitioning* the iteration space of a given loop nest into *chunks* (sets) of successive iterations

Parameter	Brief Explanation
Processor	300MHz single issue
Chip Area	$5.6mm^2$
L1 Date Cache	Private, write though, 64 lines, 4 way associative, lru replacement
L1 Instruction Cache	Private, write though, 64 lines, 4 way associative, lru replacement
L2 Unified Cache	Shared, write back, 64 lines, 4 way associative, lru replacement

Table 2.1. Architectural Details.

```
for (i=1; i<=600; i++)
  for (j=1; j<=1000; j++)
    B[i][j] = (A[i-1][j] + A[i+1][j] +
               A[i][j-1] + A[i][j+1]) / 4;
```

(a)

Slot	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	0	6	12	18	24			
2	1	7	13	19	25			
3	2	8	14	20	26			
4	3	9	15	21	27			
5	4	10	16	22	28			
6	5	11	17	23	29			

(b)

```
for (i=k*120+1; i<=(k+1)*120; i++)
  for (j=1; j<=1000; j++)
    B[i][j] = (A[i-1][j] + A[i+1][j] +
               A[i][j-1] + A[i][j+1]) / 4;
```

(c)

Figure 2.1. Jacobi computation and a possible schedule.

and *scheduling* the iteration chunks into appropriate processors. The resultant parallelized code can be represented using a *schedule table*. Figure 2.1 illustrates an example. Figure 2.1(a) gives the loop nest for the Jacobi Solver computation. Each iteration of this loop nest can be represented using a two-entry vector $(i \quad j)^T$, and there are a total of 600×1000 iterations in this loop nest. Let us assume that we divide the iteration space into 30 equal size chunks with each chunk having 20×1000 iterations. Each chunk can be represented using $\mathcal{I}_k$ $(0 \leq k < 30)$, where $\mathcal{I}_k$ is defined as:

$$\mathcal{I}_k = \{(i,j)^T \mid 20 * k < i \leq 20 * k + 20 \ \& \ 1 \leq j \leq 1000\}.$$

The mapping of the iteration space to the data space of array A which is accessed

Benchmark	N1	N2	N3	N4	N5	N6	N7
3step-log	1	1	1				
adi	4	1					
btrix	1	1	1	1	1	1	3
eflux	1	2					
tsf	1	2	1	1			

Table 2.2. The optimum number of processors that generates the minimum energy consumption for each nest of each benchmark. N1, N2, ..., N7 represent the loop nests in the applications. Note that increasing the number of processors (for any nest) further increases overall energy consumption without any benefits.

by the references shown can be explained as:

$$\mathcal{D}_A = \mathcal{D}_A^1 \bigcup \mathcal{D}_A^2 \bigcup \mathcal{D}_A^3 \bigcup \mathcal{D}_A^4 \tag{2.1}$$

where,

$$\begin{aligned}
\mathcal{D}_A^1 &= \{ \quad [i,j] \rightarrow [a,b] : (a = i-1 \quad \& \quad b = j) \quad \} \\
\mathcal{D}_A^2 &= \{ \quad [i,j] \rightarrow [a,b] : (a = i+1 \quad \& \quad b = j) \quad \} \\
\mathcal{D}_A^3 &= \{ \quad [i,j] \rightarrow [a,b] : (a = i \quad \& \quad b = j-1) \quad \} \\
\mathcal{D}_A^4 &= \{ \quad [i,j] \rightarrow [a,b] : (a = i \quad \& \quad b = j+1) \quad \}
\end{aligned} \tag{2.2}$$

In this work, we use the Omega Library [27] and Presburger arithmetic [28] to represent the iteration chunks and the data set they access. Figure 2.1(b) gives a possible schedule for the 30 iteration chunks on 5 out of 8 processors. If iteration chunk $\mathcal{I}_k$ is scheduled to be executed on processor P_n at time slot m, the corresponding table entry is k. An entry is left empty if the processor is idle at that time slot. For example, $\mathcal{I}_9$ is scheduled on P_1 at time slot 4, and processors P_5, P_6, and P_7 are idle during the entire execution. Figure 2.1(b) represents in a sense a parallelized version of the original loop nest in the form a schedule table, and the corresponding code to be executed on processor P_k ($0 \leq k \leq 4$) is given in Figure 2.1(c).

Prior study [24] shows that, in many cases, using only a subset of the available processors generates the best result in terms of performance and energy. That is, increasing the number of processors used can degrade performance and/or can increase energy consumption in some circumstances. Such a scenario may hap-

pen due to parallelization overheads, data dependencies, synchronization, etc. Table 2.2 presents the optimum number of processors for each nest of each benchmark code used in our experiments (the information on our benchmarks is given in Section 2.4.1). In our discussion of the different scheduling algorithms, we will stick to such predetermined *active processor count limit*, referred to as A in the rest of the chapter. In other words, if the active processor count limit for a loop nest is A, we use no more than A processors at each time slot in all the schedules obtained from our scheduling algorithms.

Due to the reducing scale of technology, the power density of chips has increased and the size of the heat sink that dissipates this power has decreased. As a processor expends power, the increased power density causes a rise in the operating temperature of the processor. As the operating temperature of the processor rises, it expends more leakage energy which in turn causes the temperature to rise even further. This vicious cycle of increased energy causing increased temperature which in turn causes increased leakage energy is called *thermal runaway* [13].

The thermal runaway phenomenon can result in a processor operating above a safe threshold temperature, which could in turn lead to the burning and permanent damage of the chip. In order to prevent thermal runaway, the chip should never be allowed to exceed the threshold temperature. This is called the *threshold temperature constraint*. There are many runtime techniques [13, 12] that deal with preventing thermal runaway. Our work is a compiler driven approach that determines a schedule at compile time that will never cause a thermal runaway to occur. To prevent the chip from operating at a super threshold temperature, we need to let the processor be idle periodically so that it cools down. However, this should be performed in a performance-sensitive manner.

To calculate the rise and fall of the temperature of a processor, we use the following framework which is based on the Hot-Spot tool [26]. Assuming that a processor's temperature at cycle c is T_c, a processor's temperature after running for δ cycles can be estimated using:

$$T_{c+\delta} = \mathrm{F}(T_c, \quad \delta, \quad Computation_\delta, \quad floorplan, \quad power) \tag{2.3}$$

That is, a processor's temperature at a specific time is determined by its previous

temperature (T_c), the power it expends in δ, the number of cycles during which it expends this power as well as the physical layout and properties of the processors.

The granularity of our scheduling and hence thermal model is an iteration chunk, i.e., we estimate a processor's temperature at the end of one time slot (the time for running an iteration chunk) based on its temperature at the end of the previous time slot and its power consumption and computation during the current time slot. The power consumption is a function of the computation performed by the processor in the current time slot. Since all the iteration chunks are of the equal size, we assume that the computation to execute each iteration chunk is the same. The time and energy required for executing a chunk is determined through profiling.

As a result, at compile time, we are aware of the computation chunks, the time they take to execute, and the power each chunk expends. Furthermore, the floor plan of the system is constant. Hence, given a starting temperature, our compiler can estimate the temperature changes due to the execution of the application statically. In order to do so, a scheduler based around the Hot-Spot tool was created. Hot-Spot is used to return the temperature of a processor at the end of each iteration chunk. This information is used to decide in the scheduler whether an iteration can be scheduled to that particular processor or not. To prevent a processor from going into the thermal runaway, our scheme lets a processor be idle periodically. This allows the processor to cool down so that it does not exceeding a threshold temperature. Computation is scheduled on the processor once its temperature has fallen sufficiently.

Figure 2.2 illustrates our scheme. In the first phase, the code is profiled and information such as the number of cycles it takes to execute, the size of iteration chunks, and the energy they consume are extracted. In the second phase, this information along with the details of the architecture being simulated are fed as input to our Hot-Spot based scheduler, which returns a temperature-sensitive schedule. Next, in the third phase, this schedule is optimized for data cache locality. Finally in Phase 4, the Omega Library is used to generate code based on the schedule.

Figure 2.3 presents a sample application of this scheme. The original schedule is given in Figure 2.3(a), with each black box representing an iteration chunk. For

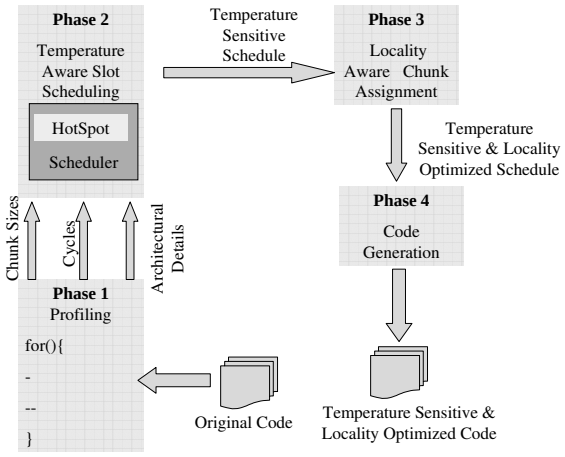


Figure 2.2. Overview of our methodology.

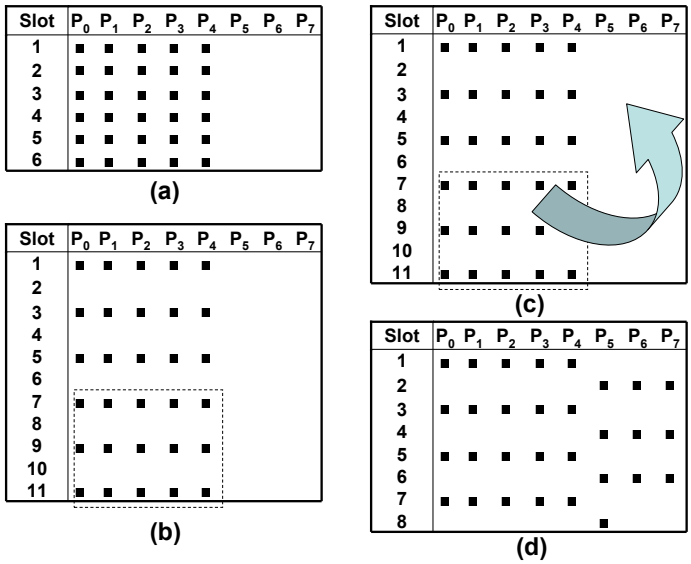


Figure 2.3. Temperature-sensitive scheduling.

the purpose of illustration, we assume the following simple temperature estimation function for each processor p:

$$T_{c+1}(p) = \begin{cases} T_c(p) + 1 & \text{if } p \text{ runs at time slot } c; \\ T_c(p) - 1 & \text{if } p \text{ is idle at time slot } c. \end{cases} \tag{2.4}$$

That is, a processor's temperature increases by 1 after executing an iteration chunk,

and its temperature decreases by 1 after being idle for a time slot. We also use this temperature estimation function in all the examples discussed further in this chapter. Further, we assume that the initial temperature is 0 and the threshold temperature is just above 1. Note, however, that we use Equation 2.3 and the real temperature scale in our experimental evaluations. The simple temperature estimation function and the "unreal" temperature numbers used in our examples are just for ease of illustration. Figure 2.3(b) gives a sample temperature-oriented schedule that satisfies the temperature constraint mentioned above. This schedule is obtained by inserting idle time slots into the original schedule in Figure 2.3(a). Obviously, this temperature-oriented schedule is better than the original schedule in terms of thermal behavior (lower average and peak temperatures). However, we also observe from Figure 2.3(b) that the new schedule takes a longer time to finish (the part enclosed in the dashed box), which is not desirable from the performance viewpoint. We can exploit the idle processors for improving the performance of a temperature-oriented schedule. Such an idea is illustrated in Figure 2.3(c). By offloading some iteration chunks from the first five processors to the other three processors (P_5, P_6, and P_7), we can reduce the performance overhead, and the resultant schedule still satisfies the temperature constraint. Figure 2.3(d) gives a possible schedule, for our example, utilizing such opportunities. This schedule needs two extra time slots as compared to the original (temperature-insensitive) schedule in Figure 2.3(a). In comparison, the schedule in Figure 2.3(b) requires five extra time slots to finish. As mentioned earlier, in all our schedules, we keep the number of active processors at a given slot the same as the original schedule. That is, if the original temperature-insensitive schedule uses at most A processors concurrently, the temperature-oriented schedule should also use no more than A processors at any slot. Note that the schedule given in Figure 2.3(d) satisfies this requirement.

2.3.2 Temperature-Driven Loop Scheduling

Let us now discuss an approach to temperature-driven loop scheduling, assuming that the active processor count limit is A. Algorithm 1 gives a sketch of the algorithm for the temperature-driven loop scheduling approach. In this algorithm,

Algorithm 1 *Temperature Driven Scheduling*

```
1: TimeSlot ← 0;
2: while exist nonscheduled iteration chunk do
3:    for all processors do
4:       P ← the A coolest processors;
5:       delete from P the processors with threshold temperature;
6:    end for
7:    schedule an iteration chunk for each processor in P at TimeSlot;
8:    remove these iteration sets;
9:    TimeSlot ← TimeSlot + 1;
10: end while
```

Slot	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
1	0	1	2	3	4			
2	5	6				7	8	9
3			10	11	12	13	14	
4	15	16	17	18				19
5					20	21	22	23
6	24	25	26	27	28			
7						29		

Figure 2.4. Example schedule using a temperature-driven loop scheduling approach.

we schedule the iteration chunks one time slot at a time. At each step, we select the coolest (in terms of temperature) processors, and the number of such processors does not exceed A. We estimate a processor's temperature from its temperature and activity in the previous time slot using the temperature estimation function (Equation 2.3). By selecting the coolest processors at each time slot, we reduce the overall temperature. It is possible that there are less than A schedulable iteration chunks at some time slots because of the threshold temperature. We repeat this process until all the iteration sets have been scheduled.

Figure 2.4 gives an application of this algorithm to the example program given in Figure 2.1, assuming that the value of A is 5. We use the temperature estimation function given in Equation 2.4. We assume further that the initial temperature is 0, and the threshold temperature is 2. For time slot 1, all the eight processors are cool, so we assign the first five iteration chunks ($\mathcal{I}_0$ to $\mathcal{I}_4$) to processors P_0 to P_4, respectively. At time slot 2, P_5, P_6, and P_7 are the coolest processors, and they are selected for time slot 2. We select P_0 and P_1 from the remaining processors

(all of them have the same temperature) to execute at this time slot. At time slot 3, processors P_0 and P_1 cannot be selected due to the threshold temperature constraint. Processors P_2, P_3, and P_4 are the coolest ones, and thus are selected. In addition, P_5 and P_6 are also selected since they satisfy the threshold temperature constraint. After all iteration chunks have been scheduled, we obtain the schedule shown in Figure 2.4. This temperature-driven schedule has better thermal behavior than the original one (in terms of both the average and peak temperatures), and it uses only one more time slot than the original schedule. However, an important problem with this schedule is that it does *not* consider data locality. By looking at the program given in Figure 2.1, one can see that data reuse happens between the neighboring iteration chunks. In the original schedule, data locality is optimized since most of the iteration chunk pairs that share data are scheduled on the same processor successively. But the schedule given in Figure 2.4 does not have this good locality behavior. Actually, none of the iteration chunks scheduled on processor P_4 have data reuse among them. The degraded data locality in this temperature-driven schedule is mainly due to the fact that we did not consider any data reuse when we schedule the iteration chunks. Note that this loss of data locality typically causes an increase in the energy expended by the processor since data has to now be moved into the L1 cache. This in turn causes the temperature to rise, which causes some extra energy to be expended the next time the data needs to be fetched. This is another example of a vicious cycle. In the next subsection, we discuss a data locality-driven scheduling approach.

2.3.3 Data Locality-Driven Loop Scheduling

Algorithm 2 gives a sketch of our locality-driven loop scheduling algorithm. This algorithm is locality-driven since, at each step, it tries to determine the chunk-processor pair, ($\mathcal{I}$, p), that is the best in terms of data reuse (lines 8–11 in Algorithm 2). In our work, the potential data reuse achieved by scheduling iteration chunk $\mathcal{I}$ on processor p is obtained by calculating the intersection set of the data elements accessed by $\mathcal{I}$ (calculated using a mapping similar to Equation 2.1), and those accessed by the last-scheduled iteration chunk on p. That is, if the set of data elements accessed by $\mathcal{I}$ is $\mathcal{D}_1$, and the set of data elements accessed by

Algorithm 2 *Data Locality Driven Scheduling*

```
1: ∀ processor p: NextSlot[p] ← 0;
2: Bound ← ⌈(number of iteration chunks)/A⌉;
3: while exist nonscheduled iteration chunk do
4:    if none of the processors is schedulable then
5:       increase Bound by ⌈(number of nonscheduled iteration chunks)/A⌉;
6:       for each processor p, p becomes schedulable if NextSlot[p] < Bound;
7:    end if
8:    for all nonscheduled iteration chunks and all schedulable processors do
9:       find the best iteration chunk and processor pair (I, p) such that data reuse is
         maximum
10:   end for
11:   schedule iteration chunk I on processor p at time slot NextSlot[p];
12:   update NextSlot[] for all processors;
13:   if NextSlot[p] ≥ Bound then
14:      p is not schedulable;
15:   end if
16: end while
```

the last-scheduled iteration chunk on p is $\mathcal{D}_2$, the data reuse is determined by the size of set $\mathcal{D}_1 \cap \mathcal{D}_2$. As has been discussed in Section 2.3.1, in this work, the data elements accessed by an iteration chunk are obtained using the Omega Library tool.

Note that we still need to make sure that the resultant schedule satisfies the active processor count limit and threshold temperature constraint. The array $NextSlot$ in Algorithm 2 captures such requirement. Specifically, for each processor p, $NextSlot[p]$ gives the next available time slot to which we can schedule an iteration chunk without violating the active processor count limit and the threshold temperature constraint. After a new iteration chunk is scheduled, $NextSlot$ is updated to reflect the change in the schedule (line 14 in Algorithm 2).

It might happen under this scheduling algorithm that some processors have many more iteration chunks scheduled on them than other processors, since a pure data locality-driven loop scheduling algorithm does not guarantee load balance among the processors. Such a scenario should be avoided since the total number of time slots to finish all the iteration chunks in this scenario can be much higher than that of a load-balanced schedule. We use the term $Bound$ in Algorithm 2 to prevent such a scenario from happening. Specifically, a processor is

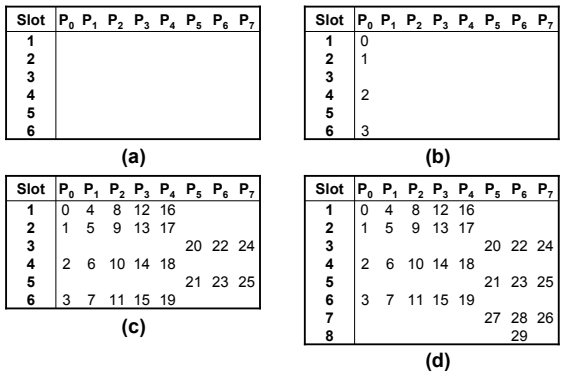


Figure 2.5. Example application of our data locality-driven scheduling approach.

not considered for scheduling the remaining iteration chunks if its next schedulable time slot exceeds *Bound* (lines 15–16 in Algorithm 2). *Bound* is initialized to $\lceil$(number of iteration chunks)$/A\rceil$ (in line 2 of the algorithm) since this is the minimum number of time slots required to finish all the iteration chunks under an active processor count limit A. When none of the processors is schedulable, but there still exist unscheduled iteration chunks, we extend *Bound* so that some processors could become schedulable (lines 4–6 in Algorithm 2). We choose $\lceil$(number of iteration chunks)$/A\rceil$ as the value to extend *Bound* since this is the minimum number of time slots required to finish all the remaining nonscheduled iteration chunks.

Figure 2.5 gives an application of our data locality-driven loop scheduling algorithm to the code shown in Figure 2.1, assuming that the value of A is 5. We use the temperature estimation function given in Equation 2.4. We assume further that the initial temperature is 0, and the threshold temperature is 2. Figure 2.5(a) gives the initial state of the schedule table. *Bound* is set to 6 as indicated by the boundary of the schedule table. Figure 2.5(b) presents the schedule after the first four iterations of the while loop in Algorithm 2. After $\mathcal{I}_0$ is scheduled on processor P_0, $\mathcal{I}_1$ is also scheduled on processor P_0, since the neighboring iteration chunks exhibit data reuse. Similarly, $\mathcal{I}_2$ and $\mathcal{I}_3$ are scheduled on processor P_0. The empty slots 3 and 5 are due to the threshold temperature constraint. Figure 2.5(c) gives the schedule after we have scheduled 26 iteration chunks. At this point, none of the processors are schedulable. Consequently, we have to extend *Bound* by 1 (to 7) since there are four unscheduled iteration chunks and $\lceil 4/5\rceil = 1$. Now, processors

P_5, P_6, and P_7 become schedulable. Iteration chunk $\mathcal{I}_{26}$ is scheduled on P_7 because it has the maximum data reuse on that processor. After we schedule $\mathcal{I}_{27}$ and $\mathcal{I}_{28}$ on P_5 and P_6, respectively, we have to extend *Bound* again by 1 ($\lceil 1/5 \rceil = 1$). After this, all the processors become schedulable, and $\mathcal{I}_{29}$ is scheduled on processor P_6 for achieving maximum data reuse.

As one can observe from Figure 2.5(d), Algorithm 2 can generate a schedule that exhibits much better data locality than the schedule in Figure 2.4 generated by the temperature-driven algorithm (Algorithm 2). However, we also observe that the schedule given in Figure 2.5(d) does not have as good load balance as the one in Figure 2.4, and the former requires one more time slot to finish. This behavior is clearly undesirable in a parallel execution environment. Therefore, we can conclude that these two scheduling algorithms have their own advantages and disadvantages. In the next subsection, we propose a loop scheduling algorithm that combines these two algorithms and has the advantages of both of them.

2.3.4 A Combined Approach to Scheduling

The idea behind our combined approach is simple. Since the temperature-driven approach is good in terms of load balancing, we first use the temperature-driven approach to determine for each processor its state (i.e., running or idle) at each time slot. After that, we use a locality-driven approach to determine the iteration chunks to be executed on each processor so that data locality is optimized. Algorithm 3 gives the algorithm for our combined approach. In the first part of this algorithm (lines 1–11), we determine the time slots at which each processor should be running an iteration chunk. This information is exploited in the second part of the algorithm (lines 12–24) to determine the next schedulable time slot for each processor. In the second part, a data locality-driven approach is used to assign the iteration chunks to each processor's running time slots for data locality optimization. Note that determining *NextSlot* and schedulable processors is much simpler in Algorithm 3 compared to Algorithm 2, since we can utilize the information from a temperature-driven scheduling. We can conclude from the above discussion that Algorithm 3 combines the advantages of both a temperature-driven algorithm and a locality-driven algorithm. Therefore, this combined approach is

Algorithm 3 *Algorithm for the Combined Approach*

```
1: mark all processors as idle for all time slots;
2: TimeSlot ← 0;
3: while exist nonscheduled iteration chunk do
4:    for all processors do
5:       P ← the A coolest processors;
6:       delete from P the processors with threshold temperature;
7:    end for
8:    mark each processor in P as running at TimeSlot;
9:    remove these iteration sets;
10:   TimeSlot ← TimeSlot + 1;
11: end while
12: ∀ processor p: NextSlot[p] ← the first running time slot for p;
13: while exist nonscheduled iteration chunk do
14:   for all nonscheduled iteration chunks and all schedulable processors do
15:      find the best iteration chunk and processor pair (I, p) such that data reuse is
         maximum
16:   end for
17:   schedule iteration chunk I on processor p at time slot NextSlot[p];
18:   NextSlot[p] ← the next running time slot for p;
19:   if no next running time slot for p then
20:      p is not schedulable;
21:   end if
22: end while
```

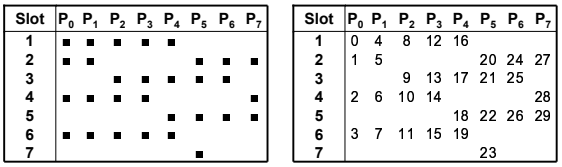


Figure 2.6. Example application of our combined approach.

expected to perform well in terms of both load-balancing and data locality.

Figure 2.6 gives a sample application of this combined approach to the code presented in Figure 2.1, assuming that the value of A is 5. We use the temperature estimation function given in Equation 2.4. We assume further that the initial temperature is 0, and the threshold temperature is 2. We first use the temperature-driven approach to obtain the active time slots for each processor. The results are given in Figure 2.6(a). Note that Figure 2.6(a) is the same as Figure 2.4 except that the entries in Figure 2.6(a) are represented as black boxes (which means the corresponding iteration chunk to be executed has not been determined yet) rather

Benchmark Name	Cycles (Million)	Leakage Energy (microJoules)	Dynamic Energy (microJoules)
3step-log	1487	945036.3	949649.9
adi	438	306134.3	933416.8
btrix	1351	921742.9	880733.3
eflux	56	40716.5	40201.6
tsf	1799	1326437.5	1221564.1

Table 2.3. Benchmarks used in our experiments. All energy values are in microJoules.

than using iteration chunk numbers. After that, we apply the data locality-driven scheduling approach to the schedule table in Figure 2.6(a), and we obtain the final schedule shown in Figure 2.6(b). Compared with Figure 2.4, the schedule given in Figure 2.6(b) has much better data locality since most of the iteration chunks that have data reuse are scheduled successively (on the same processor) in Figure 2.6(b). The schedule given in Figure 2.5(d) also has good data locality, but it needs one more time slot to finish execution compared to the schedule in Figure 2.6(b).

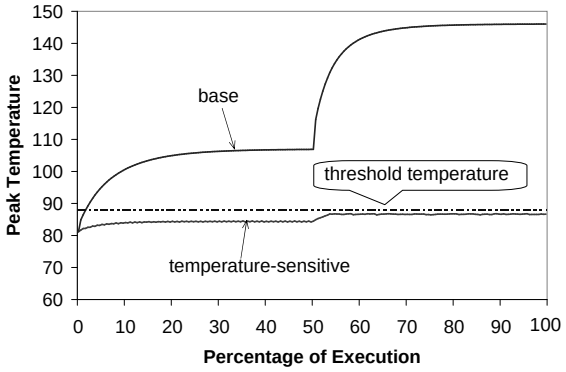
2.4 Experimental Results

2.4.1 Experimental Setup

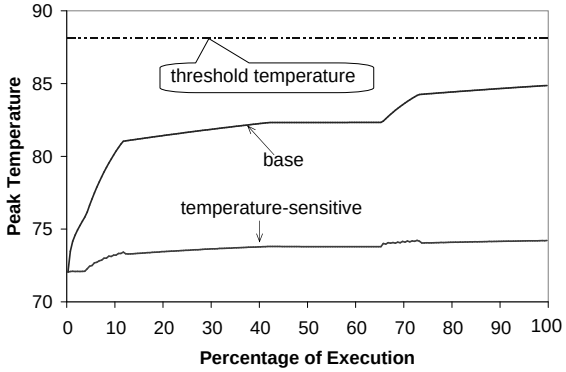
We evaluate our temperature-sensitive loop parallelization scheme using five array-intensive codes. Their important characteristics are given in Table 2.3. All the values listed in this table are obtained by executing the applications without any loop-sensitive loop scheduling. In the rest of our discussion, we refer to this version of an application as the *base version* (or the *original version*). The third column gives the number of cycles. The next two columns give the dynamic and leakage energy consumption. These values include energies consumed in processor cores, L1 and L2 caches, and off-chip main memory.

2.4.2 Experimental Results

Figure 2.7 gives the peak temperature curves for the execution of adi and eflux. The dashed lines represents the threshold temperature, which is 88.12 degrees Cel-



(a) adi.



(b) eflux.

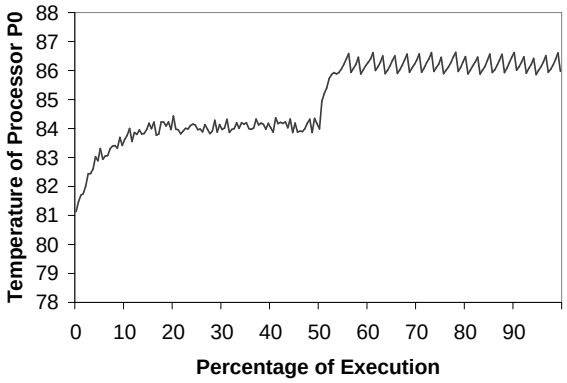
Figure 2.7. Peak temperature curves for the benchmarks adi and eflux. The peak temperature at a given time is the temperature of the hottest processor. The dashed lines represent the threshold temperature.

sius. In Figure 2.7(a), using the original temperature-insensitive parallelization strategy (referred to as "base"), the peak temperature of the processors increases and exceeds the threshold temperature shortly after the start of execution. On the other hand, after using our temperature-sensitive loop parallelization strategy (the combined approach, referred to as "temperature sensitive"), we can effectively reduce the peak temperature of the processors. Specifically, the peak temperature of the processors is always kept below the threshold temperature as can be observed from Figure 2.7(a). Figure 2.7(b) illustrates another scenario, in which the

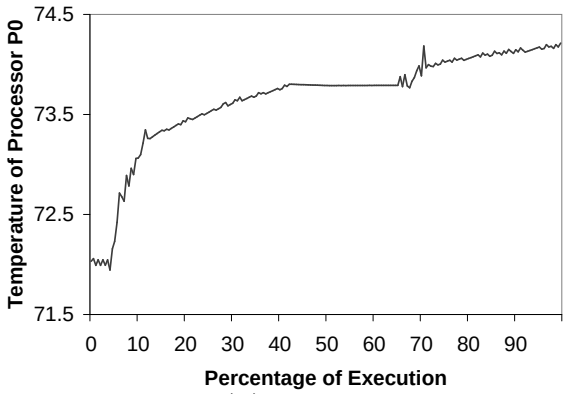
Benchmark Name	Peak Temp.		Average Temp.		Extra Energy Consumed	Extra Execution Cycles
	Original	Optimized	Original	Optimized		
3step-log	95.5	80.7	80.7	78.7	2.4%	1.8%
adi	146.1	86.8	100.5	85.0	2.4%	9.1%
btrix	84.9	78.9	74.1	73.9	0.8%	0.6%
eflux	84.9	74.2	76.4	73.7	7.4%	4.0%
tsf	87.6	74.2	80.0	73.0	1.6%	1.2%
average	99.8	78.9	81.2	76.9	2.9%	3.3%

Table 2.4. Results of our approach in terms of temperature, energy consumption, as cycles. The energy and execution cycle results are compared to the base case.

processors' peak temperature does not exceed the threshold temperature during the entire execution. Our temperature-sensitive loop scheduling scheme is still able to reduce the peak temperature of the processors, as it always selects the coolest processor for scheduling the next iteration chunk. Table 2.4 gives the experimental results of our temperature-sensitive loop parallelization approach. As we can observe from Table 2.4, our approach reduced the peak temperature and average temperature by 20.9 degrees and 4.3 degrees, respectively, when averaged over all the application tested (with respective the the base version). In addition, our approach achieves the significant temperature reduction with only a small amount of overhead in terms of energy consumption and execution cycles. Specifically, the energy consumption and execution cycles increase by 2.9% and 3.3% respectively. Figure 2.8 demonstrates how the temperature of each processor is controlled under threshold temperature. Figure 2.8(a) gives the temperature curves of processor P_0 for adi. Using the original temperature-insensitive parallelization strategy, P_0's temperature quickly goes beyond the threshold temperature. Using our temperature-sensitive approach, as processor P_0's temperatures approaches the threshold temperature, it will not be scheduled since its temperature is too high. That is, P_0 is kept idle in order for it to cool down. The decreasing slopes indicate the idle periods. Note that, during these periods, other processors that are cooler may be scheduled to execute iteration chunks. When P_0's temperature is low enough, it can execute iteration chunks (i.e., becomes active). The increasing slopes indicate the active periods. We can observe from Figure 2.8(a) that



(a) adi.



(b) eflux.

Figure 2.8. Temperature curves of processor P_0 for the benchmarks adi and eflux. The dashed lines represent the threshold temperature.

our approach can effectively prevent a processor from going beyond the threshold temperature. Figure 2.8(b) gives the temperature curves of processor P_0 for eflux. We can observe that similar trends (i.e., increasing slopes and decreasing slopes) in this figure.

In obtaining the results presented above, we set the threshold temperature parameter used in our compiler algorithm according to the physical temperature constraint, so that the peak temperature of the processors using our scheme will not exceed the physical temperature constraint. It is possible for us to lower the peak temperature further by lowering the threshold temperature parameter used in our scheme. Figure 2.9 illustrates such an example. Figure 2.9(a)-(d) give the peak temperature curves of the benchmark adi with different threshold tempera-

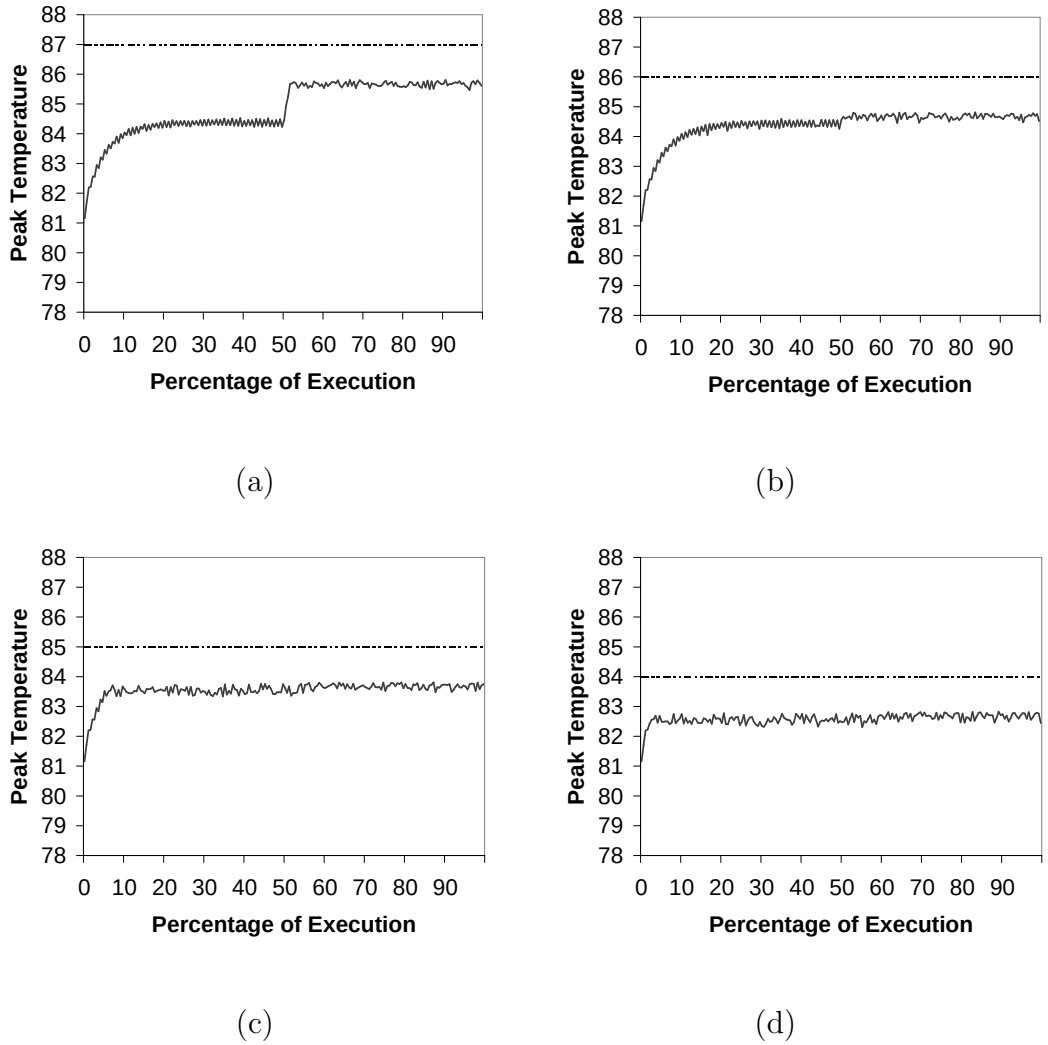


Figure 2.9. Peak temperature curves for the adi benchmark using different threshold temperature settings (87, 86, 85, and 84). The dashed lines indicate the threshold temperatures applied to obtain the corresponding curves.

tures (indicated by the dashed lines in these figures). As we can observe from the figures (together with Figure 2.7(a)), our approach is able to keep the processor peak temperature within the limit even as we lower the threshold temperature from the original 88.12 degrees to 84 degrees. We can also observe that both the peak temperature and average temperature shown in Figure 2.9(d) are lower than those shown in Figure 2.7(a). This implies that it is possible to achieve even further temperature reduction by lowering the threshold temperature parameter used in

our approach. Note that it is still meaningful in certain circumstance to reduce the peak temperature using our approach even if the original peak temperature does not exceed the threshold temperature, since a lower peak temperature may enable us to use a simpler cooling technique. However, doing so can lead to larger energy and performance overheads. There are two reasons for such overheads. The first reason is additional time slots that are required as we use lower threshold temperature in order to complete the execution. As we use lower threshold temperature, each processor needs to be idle for longer to keep its temperature below the threshold temperature, and therefore, we might need more time slots than the original one. The second reason is the extra synchronization cost due to more frequent shifting of the states of processors. If the threshold temperature is high, a processor can execute a relatively larger number of iteration chunks continuously before hitting the threshold temperature. But, with a low threshold temperature, a processor needs to be idle after executing only a small number of iteration chunks, which leads to more frequent state shifts (between active and idle) for the processors. Therefore, we need to take into account such performance overhead before lowering the threshold temperature parameter for better thermal behavior. Overall, we can conclude that our approach can achieve significant temperature reduction without incurring too much performance/power penalties.

2.5 Concluding Remarks

Many advances in computer technology have been made possible by increases in the packaging density of electronics. However, with ever-increasing levels of power consumption, power density is starting to become a serious issue before computer architects and software writers alike. As against the prior hardware-based solutions to this problem, this work focuses on compilers, and proposes three temperature-aware loop parallelization schemes for chip multiprocessors. The proposed approach takes into account temperature, load balancing, and data locality in distributing loop iterations across processors. Our experiments with five applications show that the approach reduces the maximum (resp. average) temperature by 20.9 degrees (resp. 4.3 degrees) when averaged over all the applications tested, without incurring too much performance/power penalties. Our results also show that the

temperature savings we achieve can be further improved by tuning the parameters used in our algorithm.

Chapter 3

Compiler-Directed Power Density Reduction in NoC-Based Multi-Core Designs

3.1 Introduction

High power dissipation is a very serious issue in today's microprocessors, due to increasing transistor counts and clock frequencies. One of the problems related to high power consumption is thermal emergencies [29, 15], which can be defined as the execution state at runtime where a certain temperature threshold is reached and current computation cannot continue as it is. If not tackled appropriately, thermal emergencies can lead to disastrous scenarios in terms of performance degradation and equipment loss.

Prior solutions to controlling thermal emergencies include circuit and architectural techniques such as [13, 15]. As a practical example, emergency overheating detector is a circuit technique that first appeared in the P6 series, which is also implemented in the Pentium 4, Xeon and Pentium M processors. In this technique, when a certain thermal threshold is reached, the processor suspends its execution (or powers off). While this and other similar techniques can prevent chip burnouts, frequent suspensions of execution is costly in terms of performance. This is particularly problematic in real-time systems. Also, in multiprocessor systems,

a small set of processors can easily become hotspots as a result of the workload imbalance across the processors.

This work proposes and experimentally evaluates a compiler-directed scheme that balances the computational work across the processors in an NoC based chip multiprocessor, where processors are organized as a two-dimensional mesh. The proposed compiler-directed approach makes use of ILP (integer linear programming) and operates in two phases. In the first phase, it determines the largest mesh area that can be occupied by the parallel computation without exceeding the performance degradation tolerance specified. In the second phase, it splits the workloads of select processors across multiple mesh nodes to further eliminate potential hotspots by balancing out the power density.

The advantage of using a compiler-directed scheme to alleviate the thermal emergencies is that analyzing the application provides insight on how to balance the power density optimally. This leads to reduced reliance on the hardware based schemes o prevent thermal HotSpots from occuring. This means that if there exists one hardware scheme as an absolute failsafe to throttle computation, other schemes may not be required. This represents a savings in cost for the manufacuter.

We implemented this approach and tested its behavior using a set of five applications. Our experiments reveal two important results. First, the scheme works well to reduce the occurrences of thermal emergencies. Second, the scheme also works well to improve the performance.

The rest of this chapter is organized as follows. Section 3.2 discusses related work and Section 3.3 presents the high level view of our approach. Section 3.4 gives details of the ILP approach, Section 3.5 presents our runtime results and Section 3.6 concludes the chapter.

3.2 Related Work

Thermal hotspots have been identified as an important design concern in modern processors [21, 22, 12, 25, 26]. There exist solutions to this problem based on runtime techniques involving additional hardware. Usually temperature sensors are placed in the chip, which report the temperature to an arbiter, which in turn decides whether a particular block is too hot to continue operation or whether

communication needs to be rerouted away from a potential hotspot.

Works such as [30] perform static temperature aware scheduling such that no processor in a CMP (chip multiprocessor) rises above the safe working threshold temperature. While [30] focusses on a bus-based system, out work targets NoC based architectures.

Activity migration proposed in [13] as a runtime technique reduces temperature in CMP environment by ping-ponging jobs on multiple processing elements. This method migrates computation to a different part of the chip if the temperature of one processing unit goes past a certain limit. This study does not consider the NoC design where the buffers and routers themselves can also become hot.

A runtime technique for managing thermal emergencies in the context NoCs has been presented in [15]. It proposes a throttling mechanism in order to route communication away from a potential thermal hot spot. The work described in this work is different from these prior efforts in that it does not propose changing the dynamic runtime mechanisms in the hardware that prevent chip damage. Instead, it proposes a compiler directed task mapping mechanism that reduces the number of such thermal emergency occurrences at runtime. Consequently the proposed method can work in tandem with any runtime scheme in question.

3.3 Overall Approach

In the context of NoCs, a runtime temperature control mechanism should not allow a processor or router to function normally if it approaches a temperature threshold τ.

Hence, it is important to reduce the number of such occurrences of thermal emergencies that cause shut downs. This is possible by reducing the power density otherwise known as the power per unit area of the chip. Normally, the nature of a default (i.e., performance oriented) mapping of tasks to processors is such that the communication cost between pairs of communicating processors is reduced. This however has the inverse effect of increasing the power density in a certain area of the chip (where the application is mapped), which causes performance to deteriorate (when a thermal emergency is experienced). The approach proposed in this work is to reduce the power density of the entire active area of the chip,

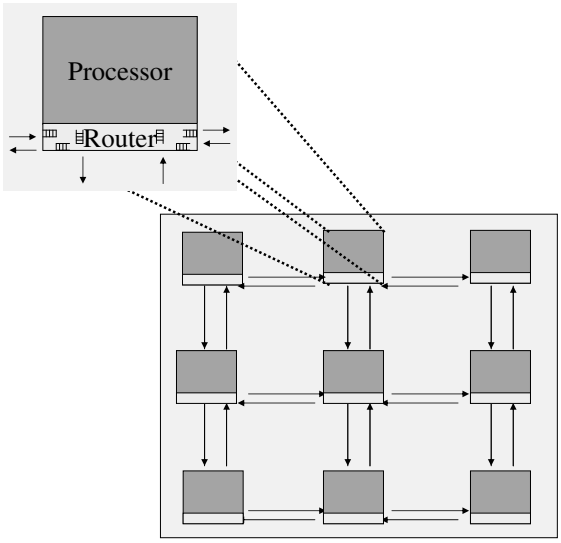


Figure 3.1. High level view of a 3*3 NoC architecture.

which is defined as the bounding rectangle formed by the active processors on the chip.

The power density can be reduced in at least two ways in the context of NoCs. First, the overall power density is reduced by increasing the size of the bounding rectangle, i.e., the rectangle in the mesh space to which the application is mapped. Second, the power density at any point within a bounding box is reduced by splitting a task scheduled originally to be executed on one processor across multiple processors. In this work, we assume that the underlying network architecture is exposed to the compiler and that the compiler can specify the task-to-processor mapping to reduce the power density.

3.4 Mathematical Programming Model

3.4.1 NoC Architecture

NoC architectures [31, 32, 33] have been proposed to overcome the problems associated with long wires used in chip wide communication. They allow a regular means of communication between on-chip computation blocks, eliminate unwanted tim-

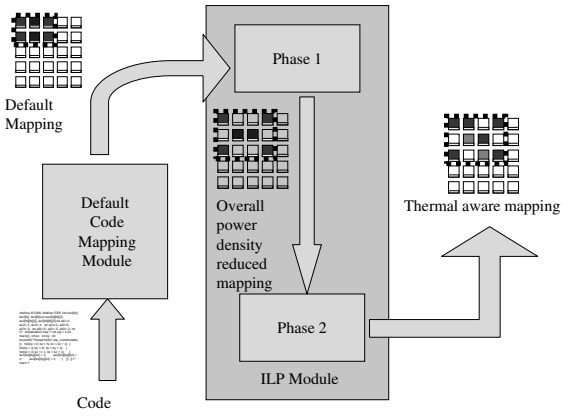


Figure 3.2. High level view of our scheme.

ing complexities, and increase the bandwidth and latency of communication [32]. Figure 3.1 shows the high level architecture of the NoC-based multiprocessor considered in this work. It comprises of a grid of processors, with a router for each processor. There exists a physical, wired connection from router to router.

The rest of this section presents the main constraints of the ILP (integer linear programming) based model proposed. Due to space constraints, the actual linear constraints are not presented; instead, their non-linear equivalents are presented. Recall that our main goal is to modify a given default (performance oriented) task-to-processor mapping to reduce the number of thermal emergencies.

The high level view of the implementation of our approach is shown in Figure 3.2. The original application is mapped onto a set of processors by the application mapping module which is provided as input to our module, which consists of two phases. In the first phase, the given task to processor mapping is modified so that the area (bounding box) occupied by active processors is increased. In the second phase, tasks that expend too much power on one processor are split and distributed among multiple processors and a new thermal aware mapping within the area calculated in the first phase is found.

Table 3.1. Terms used in the formulation.

Term	Explanation
$A_{i,j}$	Original task-to-processor mapping
$B_{i,j}$	Task-to-processor mapping generated by Phase 1
$C_{i,j,k,l}$	Original communication between $A_{i,j}$ and $A_{k,l}$
$C'_{i,jk,l}$	Communication between $B_{i,j}$ and $B_{k,l}$
$D_{i,j}$	Task-to-processor mapping generated by Phase 2
ρ	Power threshold allowed on a processor
$\mathcal{P}_{i,j}$	Input power utilization array
$R_{X_{i,j}}$	Router utilization due to horizontal communication
$R_{Y_{i,j}}$	Router utilization due to vertical communication
$R_{XY_{i,j}}$	Router utilization as a corner router
$\mathcal{T}_{i,j,k,l}$	Processor-to-processor mapping between $A_{i,j}$ and $B_{k,l}$
$\mathcal{T}'_{i,j,k,l}$	Processor-to-processor mapping between $B_{i,j}$ and $D_{k,l}$
θ	Router power
τ	Threshold temperature
W_{energy}	Total window energy
$Area$	Bounding rectangle formed by active processors
$Relaxation$	Extra communication allowance given
$\mathcal{W}$	Window size

3.4.2 Phase 1

The input to our approach is a performance oriented task-to-processor mapping as shown in Figure 3.2. This mapping is captured by the matrix A in our ILP formulation. It is assumed that processors communicate using X-Y routing. Figure 3.3 shows an example X-Y routing for communication between processors $A_{(0,0)}$ and $A_{(1,3)}$.

Number of Active processors constraints.

Constraint 3.1 below specifies that the number of active processors and the original task-to-processor mapping, A, and the new mapping, B, is the same.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} B_{i,j} \quad = \quad \sum_{i=1}^{n}\sum_{j=1}^{m} A_{i,j} \tag{3.1}$$

Graph embedding constraints.

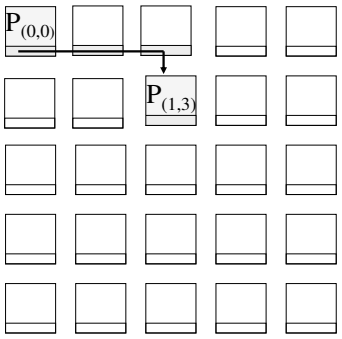


Figure 3.3. Example of X-Y routing.

Constraint 3.2 below states that the variable $\mathcal{T}_{i,j,k,l}$ is 1 if the task originally assigned to processor $A_{i,j}$ is allocated to $B_{k,l}$ in the new mapping. Constraint 3.3 ensures that each processor in B executes at most one task. Constraint 3.4 states that the number of processor-to-processor mappings between A and B is equal to the number of active processors in A.

$$\forall_{i,j} \quad \sum_{k,l} \mathcal{T}_{i,j,k,l} \quad \geq \quad A_{i,j} * B_{k,l} \tag{3.2}$$

$$\forall_{k,l} \quad \sum_{i,j} \mathcal{T}_{i,j,k,l} \quad \leq \quad 1 \tag{3.3}$$

$$\sum_{i,j,k,l,} \mathcal{T}_{i,j,k,l} \quad = \quad \sum_{i=1}^{n} \sum_{j=1}^{m} A_{i,j} \tag{3.4}$$

Communication constraints.

Constraints 3.5, 3.6, 3.7, 3.8 and 3.9 given below capture the total utilization of a router due to communication among processors. $C'_{r1,c1,r2,c2}$ is a binary matrix entry that takes the value 1 if processor $P_{r1,c1}$ communicates with processor $P_{r2,c2}$ (in the new mapping). $R_{X_{i,j}}$ and $R_{Y_{i,j}}$ give the total number of pairs of processors that use the router $R_{i,j}$ for communication in the X (horizontal) and Y (vertical) direction respectively. On the other hand, $R_{XY_{i,j}}$ counts the number of pairs of processors that use the router $R_{i,j}$ for communication in the X and the Y direction. Constraint 3.9 specifies that Sum_R is must be less than the original

communication cost plus the relaxation allowed. [1]

$$\forall_{i,j} \quad R_{X_{i,j}} = \sum_{\substack{r1,c1,r2,c2 | i=r1 \&\& \\ min(c1,c2) \leq j \leq max(c1,c2)}} C'_{\substack{r1,c1, \\ r2,c2}} \tag{3.5}$$

$$\forall_{i,j} \quad R_{Y_{i,j}} = \sum_{\substack{r1,c1,r2,c2 | j=c2 \&\& \\ min(r1,r2) \leq i \leq max(r1,r2)}} C'_{\substack{r1,c1, \\ r2,c2}} \tag{3.6}$$

$$\forall_{i,j} R_{XY_{i,j}} = \sum_{\substack{r1,c1,r2,c2 | i=r1 \&\& \\ j=c2}} C'_{r1,c1,r2,c2} \tag{3.7}$$

$$Sum\_R = \sum_{i,j} R_{X_{i,j}} + R_{Y_{i,j}} - R_{XY_{i,j}} \tag{3.8}$$

$$Sum\_R \leq Input\_R + Relaxation \tag{3.9}$$

Constraints 3.10 and 3.11 given below state that two processors can communicate only if they are both active. Constraints 3.12 and 3.13 state that a communication between two processors in the new task-to-processor mapping (B) exists if two processors in the original task-to-processor mapping (A) map on the processor in the new mapping and the original processors themselves communicated.

$$\forall_{i,j,k,l} C'_{i,j,k,l} \leq \mathcal{A}_{i,j} \tag{3.10}$$

$$\forall_{i,j,k,l} C'_{i,j,k,l} \leq \mathcal{A}_{k,l} \tag{3.11}$$

$$\forall_{\substack{i,j,k,l, \\ m,n,o,p}} \mathcal{T}_{\mathcal{C}'\ \substack{i,j,k,l, \\ m,n,o,p}} = \mathcal{T}_{i,j,m,n} * \mathcal{T}_{k,l,o,p} \tag{3.12}$$

$$\forall_{\substack{i,j,k,l, \\ m,n,o,p}} C'_{m,n,o,p} = \mathcal{T}_{\mathcal{C}'\ \substack{i,j,k,l, \\ m,n,o,p}} * C_{i,j,k,l} \tag{3.13}$$

Area constraint and objective function.

Constraint 3.14 specifies that the area of the chip occupied by the active processors. The objective funtion 3.15 maximizes the area of the chip. Note that this can increase the communication cost. However, the cost has to be kept under the sum of the old communication cost and the relaxation allowed (captured by

[1] The allowable performance degradation in this work is specified as the amount of extra communication latency permitted.

the variable *Relaxation*). Therefore, this model works towards increasing the area while keeping the communication costs under a specified limit.

$$\begin{aligned} Area \quad = \quad & \big(max(Row) - min(Row)\big) \\ & *\big(max(Col) - min(Col)\big) \end{aligned} \tag{3.14}$$

$$maximize(Area) \tag{3.15}$$

3.4.3 Phase 2

The second phase of the ILP formulation accepts as input the task-to-processor mapping B, and the communication map, C', given by Phase 1. This second phase now considers each processor as being unique based on the power it expends, which is related to the nature of the task assigned to that processor. It then splits tasks that consume too much power among multiple processors. The nature of the split is such that the area calculated by Phase 1 forms the upper bound of the area usable by Phase 2.

It needs to be noted that the very nature of splitting the computation means that the number of processors that communicate with each other increases. It is possible that the communication characteristics of this new sub-computation is different from the original computation; however, in this work, a conservative approach is adopted and all communication is assumed to be uniform. That is, if two processors, $P_{i,j}$ and $P_{k,l}$, communicate in mapping generated by phase one and if in the new mapping $P_{i,j}$ is split into $P_{m,n}$ and $P_{o,p}$; then, in the new mapping, it is assumed that $P_{m,n}$ and $P_{o,p}$ communicate with $P_{k,l}$ and no qualification about the communication is made.

Increased active processors constraint.

The splitting of the tasks that consume power above a threshold, τ, is achieved by Constraint 3.16. Here, $P_{i,j}$ is the power consumption of a processor in $B_{i,j}$. The new mapping is given by $D_{i,j}$ and the threshold power value used in splitting is ρ.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} D_{i,j} \quad = \quad \sum_{i=1}^{n}\sum_{j=1}^{m}(\mathcal{P}_{i,j}/\rho) \tag{3.16}$$

Embedding constraints.

The mapping from processors in B to those in D is captured by constraint 3.17. That is $\mathcal{T}'_{i,j,k,l}$ is 1 if a task executing on a processor $B_{i,j}$ maps onto $D_{k,l}$. Constraint 3.18 ensures that a processor in D runs only one task of B. The mapping of split processors is accounted for by constraint 3.19. According to this constraint, a task mapped onto processor in B, if split into multiple tasks, will cause that particular processor in B to be linked as many processors in D as the number of split tasks.

$$\forall_{i,j} \sum_{k,l} \mathcal{T}'_{i,j,k,l} \quad \geq \quad B_{i,j} * D_{k,l} \tag{3.17}$$

$$\forall_{k,l} \sum_{i,j} \mathcal{T}'_{i,j,k,l} \quad \leq \quad 1 \tag{3.18}$$

$$\forall_{i,j} \sum_{k,l} \mathcal{T}'_{i,j,k,l} \quad = \quad \sum_{i=1}^{n}\sum_{j=1}^{m}(\mathcal{P}_{i,j}/\rho) \tag{3.19}$$

Communication constraints.

These are similar to constraints 3.5 to 3.13 and, thus, are not presented again.

Area constraint.

This is similar to constraint 3.14 and is not presented again.

Energy window constraint.

It is ensured by constraint 3.20 below that power dissipation is spread within the active area of the chip. This is done by ensuring that, for any square window, of size $W * W$, within the chip, the total power dissipation due to processors and routers is within a preset limit W_{energy}. This has a sort of flattening effect on the power consumption withing the active area, as the area in which power is dissipated is reduced and activity is evenly spread across the active area. Here, ρ and θ are the threshold consumption of power and the power spent by one pair of communicating processors in a router, respectively.

$$\forall_{i\in 0..R-W+1, j\in 0..C-W+1} \sum_{r=i}^{i+W-1} \sum_{c=j}^{j+W-1} \Big(D(r,c) * \tau + R'(r,c) * \theta\Big) \leq W_{energy} \tag{3.20}$$

Objective function.

The objective function in this phase of our ILP formulation is to minimize the total communication cost, which ensures that the overall power density is lowered as well. This is because the total power depends on the amount of processing done (which remains constant) and the amount of communication. Since constraint 3.21 below reduces the communication, it helps to reduce the power and hence power density as well.

$$minimize(R') \tag{3.21}$$

3.4.4 Implementation Details

Our implementation in this work uses the HotSpot tool [26] in order to estimate the temperature of on-chip elements. HotSpot takes as input the floorplan of the chip, the temperature at any time i (T_i) of each element of the chip, and the power consumption of that element during a time period δ and returns the temperature of each element at time $i + \delta$. In mathematical terms, this can be represented as:

$$T_{i+\delta} \quad = \quad HS(T_i, floorplan, power, cycles, \delta)$$

In order to simulate the task-to-processor mapping, runtime processor activity and runtime processor shutdown, the algorithm shown in Algorithm 4 is used. The tasks are broken into sub-tasks called chunks. The scheduling is at the granularity of chunks. The algorithm calculates the temperature at each step and, if the temperature of a router approaches the threshold, the concerned processor is shutdown. If a router becomes too hot, then all processors that communicate using

Algorithm 4

```
1: //Time_Taken calculates the total execution time
2: Time_Taken := 0
3: while all chunks on all processors are not scheduled do
4:    Time_Taken := Time_Taken + 1
5:    T_{i+δ} = HS(T_i, floorplan, power, schedulable, cycles, δ)
6:    //the time is incremented and HotSpot function is called to estimate the temperature
7:    for each processor p do
8:       //Schedulable(p) indicates whether processor p is active
9:       //Chunk_Count(p) is the amount of computation p needs to perform
10:      if (Schedulable(p) =1) then
11:         Chunk_Count(p) := Chunk_Count(p) -1
12:      end if
13:      //If p is too hot, it cannot be active
14:      if Temperature(p) > τ then
15:         Schedulable(p) = 0
16:      else
17:         Schedulable(p) = 1
18:      end if
19:   end for
20:   //If the router is too hot, processors communicating via it, are switched off
21:   for each router r do
22:      if Temperature(r) > Threshold then
23:         for all pairs of processors (p1,p2) communicating via r do
24:            Schedulable(p1) = 0
25:            Schedulable(p2) = 0
26:         end for
27:      end if
28:   end for
29:   //Estimate the amount of communication taking place via each router r
30:   for all routers r do
31:      Schedulable(r) =0
32:   end for
33:   for all routers r do
34:      for all pairs of processors (p1,p2) communicating via r do
35:         if Schedulable(p1) = 1 && Schedulable(p2) =1) then
36:            Schedulable(r) := Schedulable(r) + 1
37:         end if
38:      end for
39:   end for
40: end while
```

that processor are shut down. The time at which the last task completes is taken as time of completion of the particular mapping.

3.4.5 Example Application of Our Approach

Figure 3.4 shows an example application of our scheme. The default (performance-oriented) mapping is given in Figure 3.4(a). The grid size is $5*5$. There are 6 tasks and they all communicate with each other. Tasks 1, 2, 3, and 4 are assumed to have power level 1. Tasks 5 and 6 are assumed to have power level 2. Hence, the average

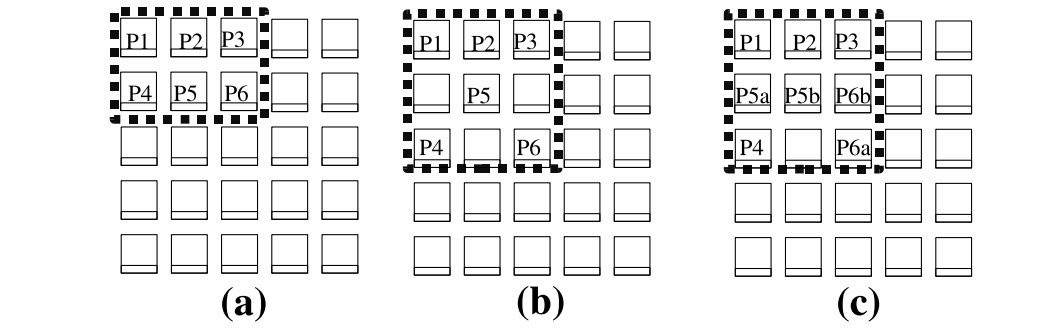


Figure 3.4. (a) Default mapping. (b) Mapping produced by the first phase. (c) Mapping produced by the second phase.

Table 3.2. Benchmarks used.

Benchmark	Cycles Millions	Processor Energy (μJ)	Router Energy (μJ)
adi	438	1239551.1	604697
eflux	56	80918.1	1696502
tsf	1799	2548001.6	515800
syntc1	438	1239551.1	0
syntc1	56	80918.1	85917071

power density is $8/6 = 1.25$ W/unit area. The threshold power level, ρ, is assumed to be 2. Figure 3.4 shows the mapping obtained at the end of the first phase of our scheme. As can be seen, the active area of the processors has increased from 6 to 9, which causes the average power density to reduce ($8/9 = .88W/unitarea$); but the power density of processors 5 and 6 is still 2. In phase 2, the workloads of processors 5 and 6 are split into 2 sub-tasks each (5_a, 5_b, 6_a, 6_b) of power level 1. This causes the average power density to remain the same, but power density at any point now is at most 1.

3.5 Experimental Results

Five loop-intensive parallel algorithms with characteristics given in Table 3.2 are used as benchmarks. The second column of this table gives the cycles for execution for each of the benchmarks and the third and fourth columns give the processor

Table 3.3. Architectural details.

Parameter	Brief Explanation
Processor	300MHz single issue
Chip Area	8.2mm * 7mm
τ	86.12 $^{\circ}C$
W	1
Mesh size	5 * 5 Grid
Processor Area	1.4mm * 1.4 mm
Router Area	.24mm * 1.4mm

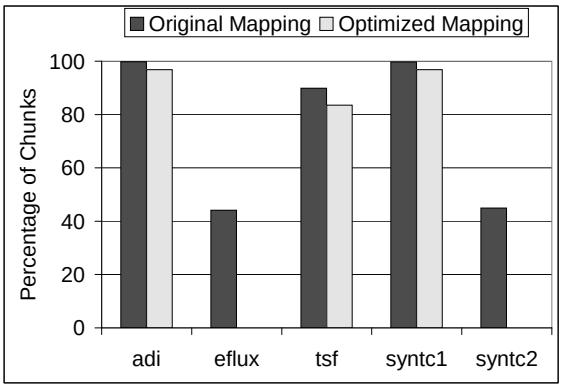


Figure 3.5. Percentage of execution chunks in which a thermal emergency is dealt with.

and router energy numbers, respectively, under the performance oriented mapping. The last two benchmarks are synthetic benchmarks that exhibit different extreme communication patterns. Specifically, in *syntc*1, very little inter-processor communication takes place, whereas in *syntc*2, the processors communicate frequently. The ILP solver used is XPressMP [34]. The details of the architecture simulated are given in Table 3.3.

Figure 3.5 shows, for each benchmark, the percentage of chunks in which a thermal emergency that requires the runtime mechanism to intervene is seen. Note that these numbers only capture the absolute number of chunks in which an emergency is seen, the number of elements affected by such an emergency is captured

by the overall performance plot given below. We see from Figure 3.5 that our thermal-aware approach cuts the the number of thermal emergencies by 42% on average.

Figure 3.6 shows the normalized performance of each of the benchmarks for the default mapping and the optimized mapping. As can be seen from this bar-chart, the proposed method reduces the overall execution time by 29% on average. This large reduction is a result of the reduction in the number of times a thermal emergency occurs. The improved performance values also capture the number of thermal emergencies that that occur in a chunk in which a thermal emergency is noted.

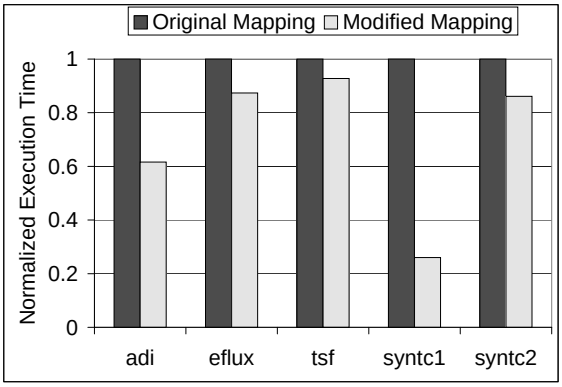


Figure 3.6. Normalized performance of the original and optimized mappings.

3.6 Conclusion

This work proposes a novel compiler-directed scheme to reduce the occurrences of thermal emergencies in NoC based chip multi-processor systems. The proposed scheme reduces the power density in the active area of the chip, which is the primary cause of runtime thermal emergencies. The scheme reduces the occurrence of thermal emergencies in all benchmarks tested and improves the overall performance

as well. The proposed scheme is implemented using ILP and works orthogonally with all dynamic, hardware based schemes that handle runtime emergencies.

Chapter 4

A Systematic Approach to Automatically Generate Multiple Semantically Equivalent Program Versions

4.1 Introduction

Design diversity is a technique used for achieving a certain degree of fault tolerance in software engineering [35, 36, 37, 38, 39]. Since exact copies of a given program cannot always improve fault tolerance, creating multiple, different copies is essential [4]. However, this is not a trivial task as independently designing different versions of the same application software can take a lot of time and resources, most of which is spent verifying that these versions are indeed semantically equivalent and they exhibit certain diversity which helps us catch design errors as much as possible (e.g., by minimizing the causes for identical errors). The problem becomes more severe if a large number of versions are required.

Automatically generating different versions of a given program can be useful in two aspects, provided that the versions generated are sufficiently diverse for catching the types of errors targeted. First, design time and cost can be dramatically reduced as a result of automation. Second, since the versions are generated auto-

matically, we can be sure that they are semantically equivalent save for the errors of interest. However, as mentioned earlier, these versions should be sufficiently different from each other, depending on the types of errors targeted.

Numerical applications which make extensive use of arrays and nested loops are good candidates for automatic version generation as they are amenable to be analyzed and restructured by optimizing compilers. Current compilers restructure such applications to optimize data locality and improving loop-level parallelism as well as for other reasons [40, 41, 42, 43]. The main stumbling block to full fledged re-ordering of computations are data dependences in the program code.

The main contribution of this work is a tool that generates different versions of a numerical application automatically *a priori*. The tool generates these versions by restructuring the given application code in a systematic fashion using the concept of *data tiles*. A data tile is a portion of an array which may be manipulated by the application. Hence, an array can be thought of as a series of data tiles. Given such a series of data tiles, of a particular size and shape, we can generate a new version of the code by restructuring the code in such a fashion that the accesses to each tile are completed before moving to the next tile. As a result, computations are performed on a per tile basis. Therefore, a different tile shape or a different order of tiles (to the extent allowed by data) gives an entirely different version of the application, thereby contributing to diversity. In this work, we also present a method for selecting the tile shapes as well as method to systematically reorder them based on the number of versions required.

We apply our tool to the emergent architectural challenge of soft errors. Soft errors are a form of transient errors that occur when charged neutrons strike logic devices which hold charges to indicate the bit that they represent [17, 18, 19, 20]. A neutron strike can change the charge held on the device either by charging or discharging it. This change in charge can lead to a bit flip in memory or logic components of the system which can affect the end results generated by the application. We show how the tool can be used to detect errors that remain undetected by a state of the art architectural recovery approach.

The remainder of this chapter is organized as follows. Section 4.2 presents the theory behind the proposed approach. Section 4.3 presents implementation details of our tool as well as results obtained using a scientific benchmark. Section 4.4

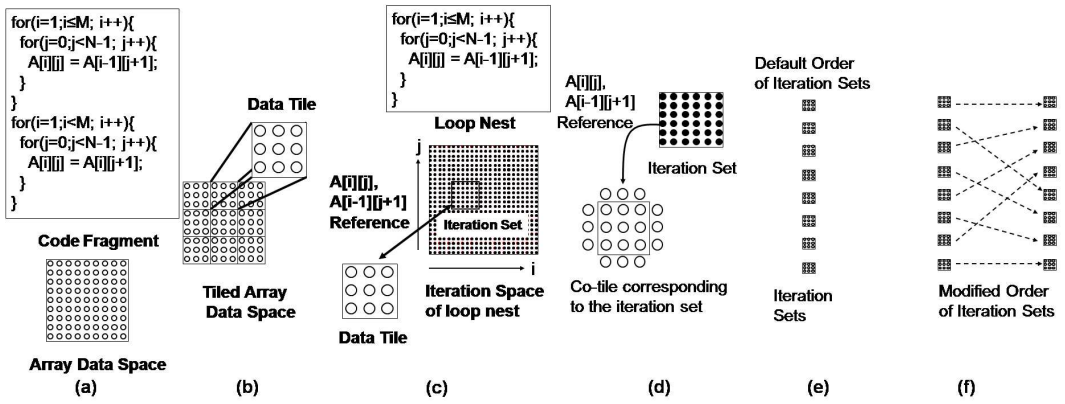


Figure 4.1. (a) A code fragment. (b) Data tiles formed from a seed tile. (c) Iteration set that accesses the data in a data tile. (d) Co-tile identification. (e) Default order of iteration sets. (f) New order of iteration sets, as a result of restructuring.

concludes the chapter by summarizing our major contributions and giving a brief outline of the planned future work.

4.2 Detailed Analysis

This section explains the details of the approach proposed to automatically create the multiple versions of a given array/loop based application. Our goal is to obtain different (but semantically equivalent) versions of a given code fragment by restructuring the fragment based on a data tile shape. The input to our approach is a code fragment that consists of the series of loop nests and the data array(s) that is accessed in the fragment. The loop nests in the fragment contain expressions, called array references, that access locations within the array. Figure 4.1(a) shows an example code fragment and the array being accessed in the loop nests.

Our approach first creates a *seed tile* which is a uniquely shaped subsection of the array (selection of a seed tile is detailed in Section 4.2.7). Using this seed tile as a template, we logically divide the array into multiple sections called *data tiles* as shown in Figure 4.1(b). In the following paragraphs we discuss what is performed on a particular data tile.

In the next stage shown in 4.1(c), we identify for each loop nest the array references that accesses locations within the data tile. Then, for each loop nest, we use these references to determine the set of iterations that access this particular

data tile. The iterations from a loop nest that are associated with a particular data tile are called the *iteration set* of that data tile with respect to that loop nest.

Now, let us consider the case for a particular iteration set associated with a data tile. It is possible that these iterations access array locations outside the data tile as well. These external locations are called the *extra tile*, and the original data tile and the extra tile are collectively referred to as the *co-tile*. Figure 4.1(d) shows the co-tile corresponding to an iteration set.

Our idea is to first identify, for each combination of data tile and loop nest, the associated iteration set. Once we have the iteration set corresponding to a data tile and loop nest, we can execute all the computations that should take place on that pair. The original code can therefore be thought of as the default order of iteration sets shown in Figure 4.1(e). Next, in order to create new codes, we systematically re-order the iteration sets to create multiple different sequences as shown in Figure 4.1(f). Each unique order of iteration sets leads to a unique version of the code. Such a re-ordering is legal provided that data dependences do not exist between iteration sets. Data dependences, impose an ordering constraint on the iteration sets and prevent full fledged re-ordering. If dependences do exist between the iteration sets, we explore other data tile shapes to arrive at a dependence free group of iteration sets.

The rest of this section details our approach. After presenting basic definitions in Section 4.2.1, Section 4.2.2 presents our method of forming data tiles. Section 4.2.3 shows how iteration sets and co-tiles are calculated. Our algorithm to detect dependences (legality requirements) are presented in Section 4.2.4. Section 4.2.5 shows how the iteration sets are systematically re-ordered. Section 4.2.6 presents the overall algorithm used to create multiple versions of code. Section 4.2.7 discusses how data tiles of different shapes and sizes are created, and Section 4.2.8 explains how we deal with code that accesses multiple arrays.

4.2.1 Basic Definitions

This subsection presents important definitions that we use to formalize our approach.

- Program : A program source code fragment is represented as $\mathcal{P} = \{\mathcal{N}, \mathcal{A}\}$, where $\mathcal{N}$ is a list of loop nests and $\mathcal{A}$ is the set of arrays declared in $\mathcal{P}$ that are accessed in $\mathcal{N}$. Figure 4.2 shows the benchmark source code fragment employed.
- Array : An array $\mathcal{A}_a$ is described by its dimensions, δ, and the extent (size) in each dimension, γ, $\mathcal{A}_a = \{\delta, \gamma\}$. For example, the array DW defined in the code fragment in Figure 4.2 can be expressed as $DW = \{3, \{10, 10, 4\}\}$ in our framework.
- Loop Nest : A loop nest $\mathcal{N}_i$, is represented as $\{\alpha, \mathcal{A}_\mathcal{N}, \vec{\mathcal{I}}, \vec{\mathcal{L}}, \vec{\mathcal{U}}, \vec{\mathcal{S}}, \psi\}$, where α is the number of loops in the nest and $\vec{\mathcal{L}}$, $\vec{\mathcal{U}}$, and $\vec{\mathcal{S}}$ are vectors that give, respectively, the values of the lower limit, upper limit, and the step of the loop index variables which are given in $\vec{\mathcal{I}}$. It is assumed that at compile time all the values of these vectors are known. The body of the loop nest is represented by ψ. The arrays accessed within $\mathcal{N}_i$ are represented as $\mathcal{A}_{\mathcal{N}_i}$ where $\mathcal{A}_{\mathcal{N}_i} \subseteq \mathcal{A}$, i.e., each loop nest typically accesses a subset of the arrays declared in the program code. For example, the second loop nest in Figure 4.2 can be represented as

$$N_1 \quad = \quad \left\{ 3, DW, \begin{bmatrix} N \\ J \\ I \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 10 \\ 10 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \psi \right\}.$$

- Loop Body : A loop body is made of a series of statements which use the references to the arrays $\mathcal{A}$ declared in $\mathcal{P}$. Consequently, loop body ψ can be expressed as a set of references.
- Iteration : For a loop nest, $\mathcal{N}_n$, an iteration is a particular combination of legal values that its index variables in $\vec{\mathcal{I}}$ can assume. It is expressed as $\vec{\mathcal{I}}_\sigma$, and it represents an execution of the loop body.
- Iteration Space : The iteration space of a loop nest $\mathcal{N}_i$ is the set of all iterations in the loop nest.
- Data Space : The data space of a data structure (e.g., an array) are all the individual memory locations that form the data structure in question.
- Reference : It is an element of ψ expressed as $(\psi_p^{r/w} = \{\mathcal{N}_n, \mathcal{A}_\mathcal{A}, L, \vec{o}\})$. It is an affine relation from the iteration space of a loop nest $\mathcal{N}_n = \{\alpha, \mathcal{A}_n, \vec{\mathcal{I}}, \vec{\mathcal{L}}, \vec{\mathcal{U}}, \vec{\mathcal{S}}, \psi\}$ to the data space of an array $(\mathcal{A}_a = \{\delta, \gamma\})$. From compiler theory [40], it is known that this relation can be described by $L\vec{i} + \vec{o}$ where $\vec{i}$ is a vector that

```
int DW[10][10][4];

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  DW[1][J][N] = 0;
}

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  for (I=2;I<=10;I++)
   DW[I][J][N] = DW[I][J][N]  -R*(DW[I][J][N]  -DW[I-1][J][N]);
}

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  DW[10][J][N] = T1*DW[10][J][N];
}

for (N=1;N<=4;N++) {
 for(II=3; II<= 9; II++)
  for (J=2;J<=10;J++)
   DW[II][J][N] = DW[II][J][N]  -R*(DW[II][J][N]  -DW[II+1][J][N]);
}
```

Figure 4.2. A code fragment with four loop nests and an array.

.

captures the loop indices of $\mathcal{N}$, L is a matrix of size $\delta * \alpha$, and $\vec{o}$ is an offset displacement vector. As an example, the reference $A[i+j-1][j+2]$ is represented by

$$\psi_p^{r/w} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

A reference within the body of a loop nest helps us calculate the locations of an array that the loop nest accesses. Further, a reference can be a read reference, which means that an array location is read from, or a write reference, which means that an array location is written to. This is identified by attaching a r/w superscript to the reference. Hence, $\psi_p^r(\mathcal{N}_n)$ represents the set of all array locations read by the reference in loop nest $\mathcal{N}_n$.

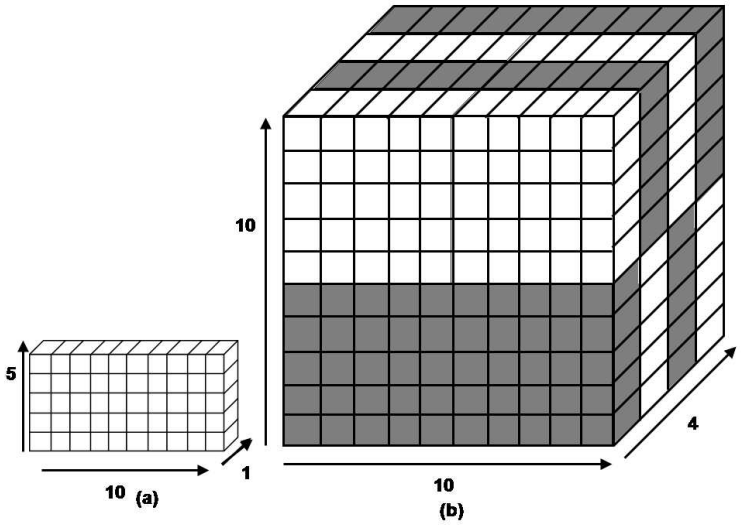


Figure 4.3. (a) Seed tile for the array DW in the code fragment of Figure 4.2. (b) The array DW divided into multiple tiles using the seed tile.

4.2.2 Data Tile Formation

In this work, we use the concept of data space tiling to logically divide the data space of an array into multiple sections. This subsection provides the theoretical basis and the algorithm used to perform tiling.

- Data Tile : A data tile $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$ is a regular subpart (region) of the array $\mathcal{A}_a$. The size of the data tile in each dimension is given by the difference between $\vec{\mathcal{L}}$ and $\vec{\mathcal{U}}$ plus 1. It is assumed that the size of a data tile is not zero in any dimension. Based on the definition of a data tile, data space of $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$ can be formally expressed as follows:

$$D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}} = \{\{d_1, d_2..d_\delta\} | \vec{\mathcal{L}}_1 \leq d_1 \leq \vec{\mathcal{U}}_1 \&\& \quad \vec{\mathcal{L}}_2 \leq d_2 \leq \vec{\mathcal{U}}_2 ... \&\& \quad \vec{\mathcal{L}}_\delta \leq d_\delta \leq \vec{\mathcal{U}}_\delta\}$$

- Seed Data Tile : A data tile, $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$, is described as a seed data tile if $\vec{\mathcal{L}} = \vec{0}$. This tile is used (as a template) to partition the array $\mathcal{A}_a$ into further tiles. As an example, Figure 4.3(a) shows a seed tile for the array DW that is defined in Figure 4.2, and Figure 4.3(b) illustrates how DW is partitioned into multiple tiles using this seed tile. This partitioning is outlined in Algorithm 1. Multiple seed tiles can simply formed by changing the values of the entries of $\vec{\mathcal{U}}$. By

Algorithm 5 $DataTile(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}})$

```
1: Tile_list := ∅
2: for i_δ = 1 to γ_δ by U[δ] do
3:    L'[δ] := i_δ
4:    U'[δ] := min(i_δ + U[δ] − 1, γ_δ)
5:    for i_{δ−1} = 1 to γ_{δ−1} by U[δ − 1] do
6:       L'[δ − 1] := i_{δ−1}
7:       U'[δ − 1] := min(i_{δ−1} + U[δ − 1] − 1, γ_{δ−1})
8:       .
9:       .
10:      for i_1 = 1 to γ_1 by U[1] do
11:         L'[1] := i_1
12:         U'[1] := min(i_1 + U[1] − 1, γ_1)
13:         Tile := D_{A_a,L',U'}
14:         Tile_list := Tile_list ⋃ Tile
15:      end for
16:   end for
17: end for
18: Return Tile_list
```

supplying different seed tiles as input to Algorithm 1, we are able to split an array into differently shaped tiles.

4.2.3 Iteration Set and Co-tile Formation

An iteration set is associated with a loop nest $\mathcal{N}_n$, and a data tile $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$. It is the subset of the iteration space of $\mathcal{N}_n$, in which the elements (iterations) have the property that $\psi_p^{r/w}(\vec{\mathcal{I}}_\sigma) \in D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$. That is, it is the set of all iterations in a particular loop nest that accesses the locations in a given data tile. We can calculate the iteration set $I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n)$ of data tile $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$ and loop nest $\mathcal{N}_n$ as

$$I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n) = \bigcup_{\psi_p^{r/w}\in\psi} \bigcup_{\mathcal{I}_\sigma\in\mathcal{N}_n} \{ \quad \mathcal{I}_\sigma \quad | \quad \{\psi_p^{r/w}(\mathcal{I}_\sigma) \cap D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}\} \neq \emptyset \quad \}. \tag{4.1}$$

Figure 4.4 shows the iteration set corresponding to the data tile of the array DW and the second loop nest in the code given in Figure 4.2. It is possible that the iteration set $I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n)$ accesses locations in the array $\mathcal{A}_a$ that lie outside

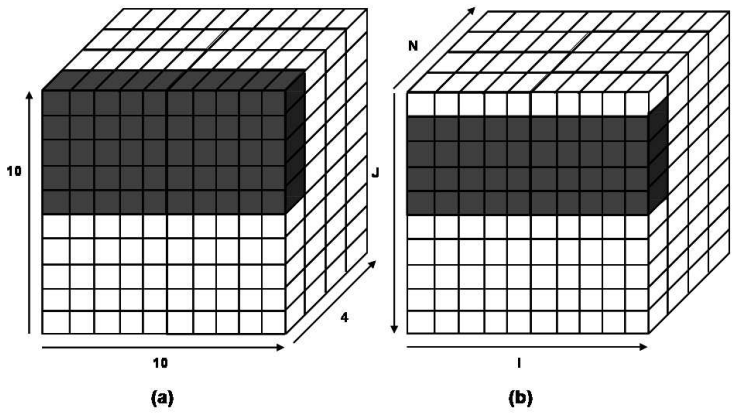


Figure 4.4. The iteration set corresponding a data tile in the array DW (accessed by the code fragment in Figure 4.2) and the second loop nest in the code fragment.

the data tile, $D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}$. In other words, $(\bigcup_{\psi_p} \psi_p(I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}, \mathcal{N}_n))) - D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}} \neq \emptyset$ may be true.

Recall that our overall goal is to capture all the computations that need to be performed by a loop nest on a data tile. As a consequence, we need to express the extra locations that are accessed by the iteration set. As mentioned earlier, the extra locations and the original data tile together are called the co-tile of the iteration set and is given by:

$$C_{D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_i} = \bigcup_{\forall \psi_p \in \mathcal{N}_n} \psi_p(I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}}, \mathcal{N}_n)) \tag{4.2}$$

Using the formulation for iteration set in Equation (4.1), the formulation for a co-tile given in Equation (4.2) and the list of all data tiles generated by Algorithm 1, we can now generate a list of all iteration set/co-tile pairs. The default list of pairs describes the default program behavior (i.e., without any restructuring). It is this behavior that we want to change while maintaining the same semantics as the original code.

4.2.4 Data Dependences Across Iteration Sets

All iterations in the given program fragment are executed in a default order called the program order. This program order can be extended to the pairs of iteration sets and co-tiles. In order to change the code, the execution of iteration sets must

Algorithm 6 *DependenceDetector*(*Tile_list*)

```
1: Dep_Array := 0
2: for all D_m ∈ Tile_list do
3:    for all N_i ∈ N do
4:       calculate I_{D_m,N_i}
5:    end for
6: end for
7: for all D_m ∈ Tile_list do
8:    for all N_i ∈ N do
9:       for all D_n ∈ Tile_list do
10:         for all N_j ∈ N do
11:            if {(⋃_{ψ^w_p} ψ^w_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^r_{p'}} ψ^r_{p'}(I_{D_n,N_j}))} ≠ ∅ ||
               {(⋃_{ψ^r_p} ψ^r_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^w_{p'}} ψ^w_{p'}(I_{D_n,N_j}))} ≠ ∅ ||
               {(⋃_{ψ^w_p} ψ^w_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^w_{p'}} ψ^w_{p'}(I_{D_n,N_j}))} ≠ ∅ then
12:               Dep_Array_{m,i,n,j} := 1
13:            end if
14:         end for
15:      end for
16:   end for
17: end for
18: Return Dep_Array
```

be re-ordered. A fundamental restriction on whether we can re-order the iteration sets are ordering relations among them, which are also known as *data dependences*.

The execution order of any two iterations can be arbitrary with respect to each other as long as these two iterations do not have any data dependence between them. A data dependence exists between two iterations within a loop nest if one iteration reads a value of a variable computed by another iteration or if both iterations compute the value of the same variable [40].

Consequently, in order to re-order any two iteration sets, there should not be any data dependence there between them. Furthermore, if we want to arbitrarily re-order all the iteration sets, there should not be any data dependence between any two iteration sets. Otherwise, it is possible that the wrong data is read by one iteration set or written by another iteration set. The rest of this sub-section presents our algorithm to detect data dependences between iteration set and co-tile pairs. This analysis is different from conventional data dependence analysis as we perform it at an iteration set and co-tile granularity.

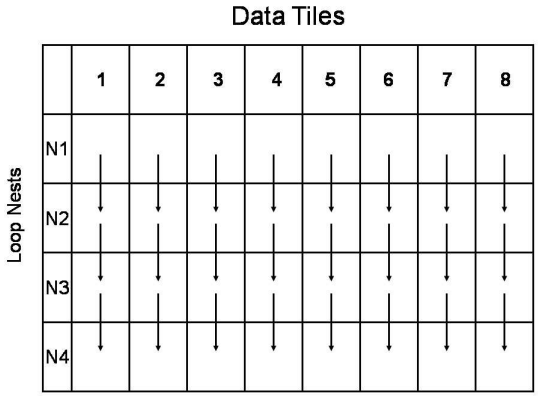


Figure 4.5. Arrows indicate the data dependence between iteration sets formed by loop nests in Figure 4.2 and data tiles formed using the seed tile in Figure 4.3(a).

Formally, two iterations $\mathcal{I}_\sigma$ and $\mathcal{I}'_\sigma$ within a nest $\mathcal{N}_n$ have a data dependence between them if and only if

$$\psi^r_p(\mathcal{I}_\sigma) = \psi^w_{p'}(\mathcal{I}'_\sigma) || \psi^w_p(\mathcal{I}_\sigma) = \psi^r_{p'}(\mathcal{I}'_\sigma) || \psi^w_p(\mathcal{I}_\sigma) = \psi^w_{p'}(\mathcal{I}'_\sigma) \tag{4.3}$$

is true, where $\psi^{r/w}_p$ and $\psi^{r/w}_{p'}$ are two references that appear in $\mathcal{N}_n$.

This formulation can be extended to iteration sets and the co-tiles that are accessed in them. In the context of our work, a dependence is said to exist between two iteration sets if and only if,

$$\begin{aligned}
&\{(\bigcup_{\psi^w_p} \psi^w_p(I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n))) \quad \cap \quad (\bigcup_{\psi^r_{p'}} \psi^r_{p'}(I(D_{\mathcal{A}_a,\vec{\mathcal{L}}',\vec{\mathcal{U}}'},\mathcal{N}_{n'})))\} \neq \emptyset || \\
&\{(\bigcup_{\psi^r_p} \psi^r_p(I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n))) \quad \cap \quad (\bigcup_{\psi^w_{p'}} \psi^w_{p'}(I(D_{\mathcal{A}_a,\vec{\mathcal{L}}',\vec{\mathcal{U}}'},\mathcal{N}_{n'})))\} \neq \emptyset || \\
&\{(\bigcup_{\psi^w_p} \psi^w_p(I(D_{\mathcal{A}_a,\vec{\mathcal{L}},\vec{\mathcal{U}}},\mathcal{N}_n))) \quad \cap \quad (\bigcup_{\psi^w_{p'}} \psi^w_{p'}(I(D_{\mathcal{A}_a,\vec{\mathcal{L}}',\vec{\mathcal{U}}'},\mathcal{N}_{n'})))\} \neq \emptyset
\end{aligned} \tag{4.4}$$

is true.

Based on Equation (4.4), Algorithm 2 detects the data dependences between the iteration sets formed from a list of data tiles. As we are not interested in re-ordering the iterations within an iteration set, dependence detection is performed at the level of loop nest granularity. The algorithm sets $Dep\_Array[m, i, n, j]$ to 1

if a dependence exists between the iteration set $I_{D_m,\mathcal{N}_i}$ and the iteration set $I_{D_n,\mathcal{N}_j}$, where D_m and D_n are data tiles created by Algorithm 1. For two iteration sets associated with the same loop nest, the dependence flows from the iteration set that contains the earlier iterations to the other iteration set. Let us now discuss what the matrix Dep_Array represents. The dependence relations between iteration sets can be described by a graph in which the nodes are the individual iteration sets. A directed edge from the node that represents iteration set $I_{D_m,\mathcal{N}_i}$ to the node that represents $I_{D_n,\mathcal{N}_j}$ means that $I_{D_n,\mathcal{N}_j}$ is dependent on $I_{D_m,\mathcal{N}_i}$. Consequently a node that represents an iteration set that is independent of all other iteration sets has a fan-in value of zero in this graph. Given these observations, we can conclude that the matrix Dep_Array is simply the representation of this graph in an adjacency matrix form. The dependence relations between iteration sets is represented pictorially in Figure 4.5.

At this point, we have generated a list of iteration sets which when executed individually perform all the computations that should be performed on a particular data tile by the associated loop nest. However, it is possible that two iteration sets, $I_{D_n,\mathcal{N}_j}$ and $I_{D_{n'},\mathcal{N}_j}$, which are associated with the same nest and have a dependence between them, might intersect. That is, some iterations may belong to both $I_{D_n,\mathcal{N}_j}$ and $I_{D_{n'},\mathcal{N}_j}$. In order to produce code that is semantically identical to the original code, the intersecting iterations need to be associated with only one of the iteration sets. Assuming that the iteration set $I_{D_{n'},\mathcal{N}_j}$ is dependent on $I_{D_n,\mathcal{N}_j}$, the intersecting iterations are executed by $I_{D_{n'},\mathcal{N}_j}$. That is, $I_{D_n,\mathcal{N}_j}$ is set to $I_{D_n,\mathcal{N}_j} - (I_{D_n,\mathcal{N}_j} \cap I_{D_{n'},\mathcal{N}_j})$.

4.2.5 Re-ordering Iteration Sets

The key requirement for full re-ordering of iteration sets is that there should be no data dependence at all between iteration sets. However, this behavior is not exhibited by most real applications. Therefore, we relax this requirement and allow reordering when the only dependences are between iteration sets corresponding to the same data tile. That is, directed edges of the form, $I_{D_n,\mathcal{N}_i}$ to $I_{D_n,\mathcal{N}_j}$ which represents data dependences between iteration sets associated with the same data tile are allowed. Once this condition has been satisfied, we first group all the iteration sets associated with each tile. Then, we partition the groups of iteration

Position	Code Version							
	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	1
3	3	4	5	6	7	8	1	2
4	4	5	6	7	8	1	2	3
5	5	6	7	8	1	2	3	4
6	6	7	8	1	2	3	4	5
7	7	8	1	2	3	4	5	6
8	8	1	2	3	4	5	6	7

Figure 4.6. The different orders of iteration sets in the different versions of the code

Algorithm 7 $VersionGenerator(P_o, V)$

```
1: Generate Seed Tile
2: Create Iteration Sets and Partitions
3: Verify dependences
4: while Dependences exist do
5:    if Generate New Seed Tile() == failure then
6:       Return Error
7:    end if
8:    Create Iteration Sets and Partitions
9:    Verify dependences
10: end while
11: Create V Versions
```

sets into V groups, where V is the number of versions of code that are required and number each partition from 1 to V. We use this numbering to create a circular sequence over all the iteration set partitions. That is, to create the i^{th} version of the code the order of iteration set partitions is : i, $i+1$ ... $V-1$, V, 1, 2....$i-2$, $i-1$. Figure 4.6 presents the orders of partitions when V is 8.

[t]

4.2.6 Generating Multiple Versions

This section describes Algorithm 3 to create the multiple versions of an input program. The input to the algorithm is the original program P_o and number of versions, V, of the code that are desired. In order to create a semantically equivalent version of P_o, a new seed element (that has not been used previously)

```
int DW[10][10][4];

for (J=2;J<=5;J++)
  DW[1][J][1] = 0;

for (J=2;J<=5;J++)
  for (I=2;I<=10;I++)
    DW[I][J][1] = DW[I][J][1]  -R*(DW[I][J][1]  -DW[I-1][J][1]);

for (J=2;J<=5;J++)
  DW[10][J][1] = T1*DW[10][J][1];

for(II=3; II<= 9; II++)
  for (J=2;J<=5;J++)
    DW[II][J][1] = DW[II][J][1]  -R*(DW[II][J][1]  -DW[II+1][J][1]);
```

Figure 4.7. The code generated for one data tile of the code given in Figure 4.2.

is formed. Then, using this seed element, the data space of P_o is broken up into further data tiles.

Using these data tiles and the loop nests in P_o, the dependence graph between the iteration sets that correspond to these data tiles is created. If there are no dependences between iteration sets corresponding to different data tiles, then the different versions of the code are created using orders as explained in Section 4.2.5. If however, dependences do exist, a new seed tile is used. If no satisfactory seed tile can be found, an error is reported. In order to generate the actual code, we rely on the Omega Library [44] which is a polyhedral tool in which iteration spaces can be described using Presburger arithmetic [45]. Given the description and order of the iteration tiles, the *codegen* utility of the Omega Library is used to generate the actual loop nests. Once the loops have been generated, they are combined so that the generated code is as compact as possible. However, the combining is done such that the order between the iteration sets remains the same. In fact, the combining method simply generates loops that iterate over the partitions of iteration sets. A portion of the semantically equivalent version of the code corresponding to one data tile is shown in Figure 4.7.

4.2.7 Data Tile Selection

So far we have ignored the problem of generating the actual seed tiles which divide the array data space into its component tiles.

The potential space to explore in order to select appropriate seed tiles is vast. We first trim this space by considering only those tiles whose boundaries are parallel to the axes of the array that is being tiled. The rationale behind this is that the output codes generated using such tiles tend to be simpler that those generated using arbitrary tiles. That is, if the array is δ-dimensional, the seed is shaped regularly, and the references from the loop nest to the array are through *affine* expressions; then iteration sets that access the data tiles are regular in shape.

Further, as we require V different versions, we assume that the size of the seed tile should imply that there are V data tiles. This also implies that the iteration sets in an iteration set partition are all associated with the same data tile.

Let us consider a δ-dimensional array, $A[n_1, n_2, ..n_\delta]$ for which V unique seed tiles are required. As A is δ-dimensional, any seed tile of A, $S[s_1, s_2, ..s_\delta]$, is also δ-dimensional. Therefore, the problem of finding the values of $s_1, s_2, ..s_\delta$ which defines the shape of the seed tile translates into the problem of selecting an appropriate value of s_i from the factors of n_i such that $\sum_i s_i = V$. As n_i is bounded by the array size, the number of combinations from which $S[s_1, s_2, ..s_\delta]$ is selected is not very large.

4.2.8 Handling Multiple Arrays

Our formulation so far has assumed that the references in the loop nests (of the code for which we meant to generate multiple versions) access a single array. In order to extend our approach to multiple arrays, we first need to extend the concept of an iteration set. An iteration set is now associated with a loop nest as well as data tiles belonging to different arrays. As a result, the iteration set is expressed as $I_{\{D\},\mathcal{N}_j}$, where $\{D\}$ is the set of data tiles (from different arrays) which are accessed in that iteration set. If the loop nest associated with the iteration set does not contain references that access an array $\{D\}$ will not contain a data tile from that array. Consequently, dependences between two iteration sets can potentially occur if they both access a common location in any array used by the program.

Another consideration with multiple arrays is how the seed tile for each array is created. One approach is to simply have the same seed tile for each array. Another approach is to create different seed tiles for different arrays, where the shape of a seed tile associated with one array is independent of the seed tile chosen for another array. In yet another approach, a seed tile is created for a chosen array $\mathcal{A}_s$ with a fixed number of elements. The ratio of the elements in a seed tile for an array $\mathcal{A}_{s'}$ is fixed relative to the number of elements in a seed tile used for $\mathcal{A}_s$, and based on the number of elements in this seed tile, the shape of the tiles is determined. Consequently, by changing the number of elements in the seed tile used for $\mathcal{A}_s$ the seed tile used for $\mathcal{A}_{s'}$ is changed. As each approach potentially gives us different versions of code, the approach we choose depends on the number of versions that need to be created. The default approach used is the one in which each tile in each array is of the same shape.

4.3 Implementation and Experiments

While our automated approach can be useful in any scenario where multiple versions of the same code are needed, we focus on one particular scenario in this work. This section first describes the targeted scenario where our proposed approach is applied. It then illustrates the architecture of the tool that is created based on the approach. Finally, it describes the experiments conducted using the tool in the targeted scenario. As mentioned earlier, soft errors are a growing threat to the correct execution of an application [17, 18, 19]. A soft error is defined as an unwanted change in the state of a bit in a computer's component such as the memory system. It can result from particle strikes on logic devices which cause the bit represented by the device to flip. Increased scaling of technology has exacerbated this problem [20]. As result, the problem of soft errors has received considerable attention with many proposed hardware as well as software solutions. In chip multiprocessor (CMP) architectures, redundant-threading (RT) is one of the ways to overcome soft errors [16]. In an RT framework the same code is simultaneously executed across all the processors and periodically the results are compared to check if the computed results across the different threads agree. If they agree, it is assumed that no error has occurred as only a single soft error is expected in

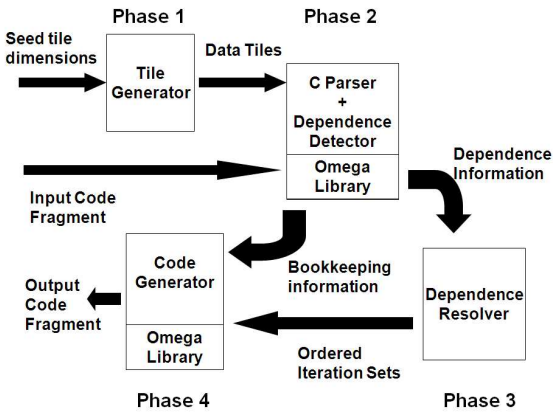


Figure 4.8. Details of the flow within the tool. Phase 1 involves the creation of data tiles (Section 4.2.2) using a unique seed tile (Section 4.2.7). Phase 2 involves the parsing of the input code fragment, formation of iteration sets (Section 4.2.3), and detection of data dependences between them (Section 4.2.4). Phase 3 re-orders iteration sets (Section 4.2.5). Finally, phase 4 generates the output code fragment using the Omega Library (Section 4.2.6).

any single thread and in any time frame. Another way is to run the code multiple times one after another and to check whether the results from each run agree with each other. Obviously, running each version simultaneously, if the resources are available, is the preferred option as it results in a faster finish time for the thread. The disadvantage is that in a CMP that is based on the shared memory concept, threads that operate simultaneously in the RT framework would read the same data from memory in close temporal proximity. Therefore, if a datum in memory is corrupted by a soft error, running the same code multiple times in parallel could result in the corrupted value being read by all threads. Such a read could result in the wrong result being computed and this error would remain undetected in current techniques. Although error correcting codes (ECC) have been proposed to overcome errors in the caches, ECC is not a viable solution in all computing systems due to the high costs it involves, especially form the power consumption angle [46, 47]. Furthermore, ECC would not catch multiple errors, which would be detected by our method.

We propose to use our automatic versioning algorithm to create multiple versions of the thread. These versions, when run in parallel, will access data in different temporal orders. Thus, the proposed approach will achieve temporal diversity without increasing the overall execution time. As a result, a particular datum

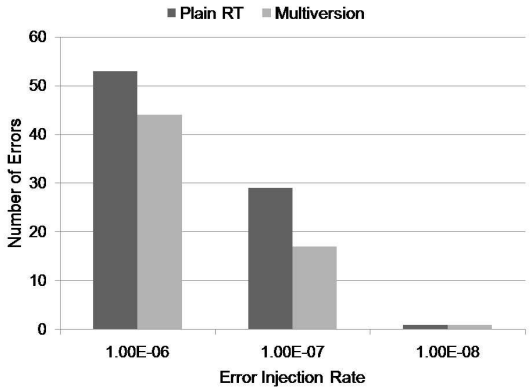


Figure 4.9. The graph shows the number of errors in the array *DW* for different error injection rates using the default RT scheme and the proposed approach.

which is corrupted at some time during the execution of the threads, could be accessed before corruption by one thread and after corruption by another. Therefore, it is possible that the changed value of the datum will be observable in the results of the different threads. Obviously, if the datum does not affect the end result, the proposed approach would perform exactly like the RT case and declare that no soft error has occurred. However, if that datum affects the end results, our approach is more likely to detect it. A tool was implemented based on the data tile based code restructuring approach (see Figure 4.8). This tools uses the Omega Library to evaluate the relations described in Section 4.2 and to generate the loops corresponding to the final relations using the Library's *codegen* utility on each relation one by one [48]. The tool was used to automatically create eight versions of the *tsf* benchmark shown in Figure 4.2 using the seed tile shown in Figure 4.3(a). Each version used a different order of iteration tiles shown in Figure 4.6. Therefore, each iteration set will execute at a particular time slot in at least one version. In an error free scenario, the different versions should generate the *same* results. However, in case of a soft error, two versions may differ in the results generated for a particular iteration set. In that case, the version that scheduled the iteration set earlier than the other is assumed to be the correct one. That is, the error is assumed to have occurred between the executions of the earlier set and set executed later.

We ran the original benchmark in conjunction with a fault injection module

[49] to simulate execution under the soft error scenario. This setup was used to record the *stage* at which each error was injected and where in the memory space it occurred. Then, each automatically generated version of the code was run under the error injection mode using the previously recorded error occurrence and the results were compared with each other. A simple arbiter is used to reason about the results that are generated. If the results of any data tile in the automatically generated versions were different, the arbiter chose the results of the version in which the iteration tile corresponding to the data tile is executed earlier. In order to simulate RT, the errors recorded earlier are injected for each version, one at a time. At each stage, any error that is not injected into the memory is assumed to be caught, but any changes to the memory itself are allowed to propagate. Figure 4.9 shows the number of remaining errors in the proposed approach as compared to the standard RT approach (which uses the same version in each processor) for different injection rates. It can be seen that the proposed approach reduces the number of errors that affect the end result.

4.4 Concluding Remarks

This work presents a tool that uses code restructuring techniques to automatically generate multiple semantically equivalent versions of a given numerical application that is organized as a series of loops that access data in arrays. We created different versions of the code that differ in the order in which they access the data and used these different versions of the code to detect the occurrence of soft errors during the execution of the code. We believe that, this tool provides an inexpensive and automated method to enable fault tolerance to critical applications. Our planned future work includes developing more techniques to generate seed tiles easily and developing techniques to generate more compact code. We also plan to use our tool in other scenarios that benefit from multiple versions.

Chapter 5

Performance Aware Secure Code Partitioning

5.1 Introduction

Many embedded as well as distributed applications exist where decisions are made using sensitive information [50, 51, 52]. The execution environment of such applications is among hosts that have varying degrees of access to data such as the one outlined in [53]. In such an environment, the application is partitioned by a compiler or splitter so as to take into account the different inter-host trust levels and data sharing patterns [52, 54]. Our focus, in this work, is on performance aware secure code partitioning among a set of hosts. The scenario targeted involves a set of hosts that want to execute a secure embedded application in parallel. The data structures manipulated by the application can exhibit very different inter-host sharing patterns. For example, while one data structure can be manipulated by all hosts, another one can be accessed by only two hosts, and so on. Another input to our approach is a host hierarchy that indicates the relationships between different hosts, i.e., whether any data that can be accessed by host h_i can also be accessed by host h_j, and so on. Our compiler-guided approach partitions the application among the hosts such that no access control violation occurs (if such a partition is possible). At the same time, we want to finish the execution of the application as quickly as possible. That is, we want to reduce the execution time

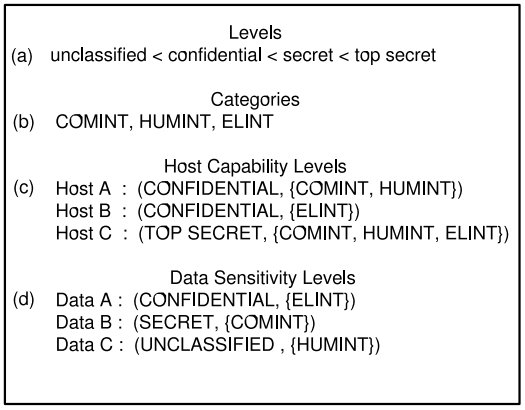


Figure 5.1. Multilevel security. (a) Lattice of different security levels (b) Categories of data (c) Host capability levels of (d) Data sensitivity levels.

without compromising data sensitivity. We achieve minimum execution time by balancing the workloads of the hosts based on a given host hierarchy. Note that reducing execution time brings two benefits. The first one and more obvious one is that the overall work is completed in a shorter period of time which implies better performance. The second benefit is that, since the program execution takes less time, the time frame during which the program is potentially vulnerable to security-related attacks is reduced.

The application of this work is in many fields including balancing the hosts in a multilevel security system (MLS). An MLS uses qualifiers on the data (objects) in a system to classify them according to their level of sensitivity and qualifiers on hosts (subjects) to separate them according to their capability level [53, 55]. It allows access to data with a certain sensitivity level to hosts with a corresponding capability level and prevents all other accesses. MLS is not a perfect solution to ensure the privacy of data [56] but has wide practical usage [57].

A classic MLS lattice is shown in Figure 5.1(a) [53]. This lattice shows that the most sensitive level is top-secret and the least sensitive is unclassified. Further, there exists a partially ordered set of categories shown in Figure 5.1(b). The categories can differ depending on the nature of the objects being protected by the MLS. The categories in this figure are several types of intelligence [58]. An example of host subjects which are assigned different clearance levels is shown in Figure 5.1(c). Data objects assigned various sensitivity levels are shown in Figure

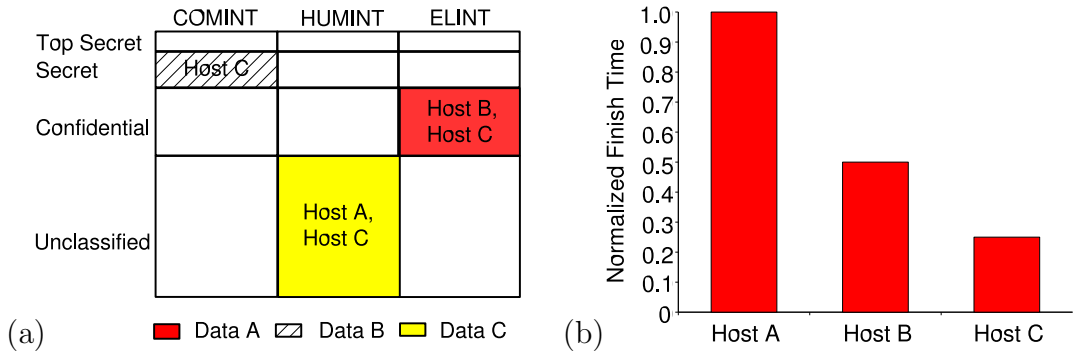


Figure 5.2. (a) Data accessible to the different hosts. (b) Normalized execution time for the hosts without load balancing.

5.1(d). A host is allowed to access a particular data if the following two conditions are met [53, 55]:

1. the sensitivity level of the host $\geq$ the sensitivity level of the data object
2. the categories of the host $\subseteq$ the categories of the object

Figure 5.2(a) shows a data structure which is divided into 12 parts. Each part corresponds to a particular category-sensitivity pair. We see that less sensitive portions are larger. The shaded portions correspond to the data objects given in Figure 5.1(d). The figure also shows that host C can access all the portions accessible by host A and host B.

Let us now consider what would happen in case a READ operation was required on all the data. In a scenario without load balancing, i.e. where sharing of data structures does not take place, host A would read data C, host B would read data A and host C would read data B. The finish times for this execution are shown in Figure 5.2(b). It can be seen that host C, which could have performed more operations had sharing been allowed, remains idle and hence the overall performance is not ideal. This motivates the possibility of balancing the loads across the hosts to achieve better performance.

This work treats the capabilities of the various hosts as ownership rights of hosts over data. If two hosts have the capability to access a particular data element, they are said to share that element. This work takes the hierarchical relation

between hosts into account to produce the partitioning of computation between these hosts in such a way so as to reduce the overall computation time without security violations. While the inter-host relationships can be expressed as a directed acyclic graph (DAG) [59]; in this work we consider the scenario in which the relationship between the hosts is described by a tree. We expect that the proposed scheme can be modified to cater to DAGs.

In the context of CMPs, this work treats each host as a thread/process executing on a CMP. Secure code parititioning a method to ensure that the confidentiality and integrity of the data is maintained. Robustness in this context would mean that no thread accesses data that it should not have access to. Performance in this context would mean that the different threads have equal workload and would therefore complete execution at roughly the same time. A partitioning scheme that only cares about performance would simply ignore the host capability and data sensitivity levels and assign equal amount of load to each thread. This scheme is very likely to lead to a loss of data confidentiality and by consequence a loss in robustness. On the other hand, a purely confidentiality oriented approach would ensure robustness but may ignore perfromance requirements. This work bridges the gap between these two extremes and makes secure code parititioning a practical approach from a performance viewpoint as well.

The rest of this chapter is structured as follows. The next section discusses our compiler-driven approach to secure code partitioning in detail. Section 5.3 presents an experimental evaluation using two representative application scenarios. Section 5.4 discusses related work, and Section 5.5 summarizes the chapter.

5.2 Code Partitioning

5.2.1 Overview

The approach proposed in this work takes two inputs in addition to the program to be partitioned: data decomposition information for each data structure and a host hierarchy (see Figure 5.3(i)). The output is a partitioned code across the hosts. The data decomposition information indicates which parts of each data structure can be accessed by which hosts. Note that, each data structure in the application

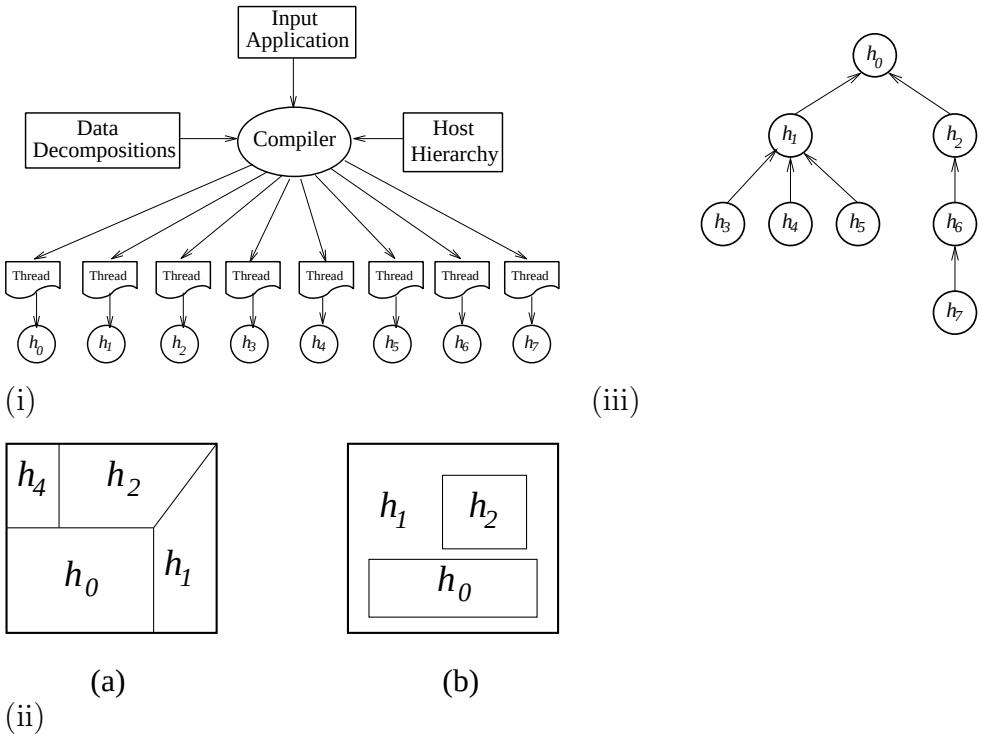


Figure 5.3. (i) Secure code partitioning. (ii) Two different data decompositions. (iii) An example host hierarchy tree (HHT).

can have a different type of decomposition and can be accessed by (potentially) different sets of hosts. For example, Figure 5.3(ii) illustrates how two different data structures (in this case, two-dimensional arrays) are decomposed between hosts. It must be emphasized that what we mean by data decomposition here is not physical partitioning of data across hosts; it just indicates the portions of the data structure that can be manipulated by different hosts. Another important point is that, when a data region is marked with a host name (say h_i), it means that this region can be accessed by h_i, or any other host h_j that has the privilege of accessing any data that can be accessed by h_i.

The host hierarchy, indicates how data can be shared among the hosts. Specifically, $h_i \rhd h_j$ means that any data that can be accessed by host h_i can also be accessed host h_j. In this case, we also say that host h_j *dominates* host h_i. Our framework uses this information to balance the workloads of the hosts to reduce program execution time. A host hierarchy can also be expressed using a *host hierarchy tree* (*HHT*). In this tree, each node represents a host, and a directed edge

from h_i to h_j indicates that $h_i \rhd h_j$. Figure 5.3(iii) shows an example HHT that involves 8 hosts. Note that, in this HHT, h_0 can access any data that can be accessed by other hosts; h_1 can access any data that can be accessed by h_3; and so on. Finally, it is important to note that, in a given decomposition, an array region can be marked by multiple host names. That is, it is possible that an array region can be marked with both h_i and h_j, where neither $h_i \rhd h_j$ nor $h_j \rhd h_i$ holds true. This allows the programmer to indicate those sharings that are not implicit in the HHT. For example, the programmer can indicate that h_3 and h_4 can access a given array region, while there is no dominance relationship between these two hosts in the HHT shown in Figure 5.3(iii).

Our goal in this work is to partition the input code across the hosts in a secure manner. Our focus is on embedded codes that are structured as a series of loops accessing array data. Consequently, our unit of workload distribution is a loop iteration, and code partitioning in our context means distributing loop iterations across the available hosts based on data space decomposition and the host hierarchy. That is, each host can only execute the loop iteration points that it is allowed to, and at the same time, we want minimize overall execution cycles. Note that, for a given application code, performance aware secure code partitioning may or may not be possible depending on data decomposition and the host hierarchy under question. However, when it is possible, our approach will return a solution. As far as minimizing execution cycles is concerned, we evaluate two metrics in this study:

1. The *Execution time* measured in terms of loop iterations of the the host that finishes its portion of iterations last.

2. The *Standard deviation* (STD) between the execution times of the hosts. STD is used to indicate how evenly the workload has been spread across the different hosts.

5.2.2 Representation of Data and Iteration Sets

To identify loop iterations that access data in a particular array region, we use Presburger formulas [60]. Presburger formulas are those formulas that can be constructed by combining affine constraints on integer variables with the logical

Table 5.1. Notations used.

Notation	Brief Explanation
h_i	host i
$\vec{I}$	a loop iteration point
$\vec{d}$	a data (array) index point
I_k	loop nest k and the set of iteration points in it
A_j	data structure (array) j
$S(I_k)$	set of data structures that are accessed by loop nest I_k
$R(A_j, I_k)$	set of references to data structure A_j in loop nest I_k
N_{avg}	average number of loop iterations per processor in the ideal case

operations $\vee$ (or), $\wedge$ (and), and $\neg$ (not), and the quantifiers $\exists$ (existential) and $\forall$ (universal). The affine constraints can be either equality or inequality constraints.

Table 5.1 gives some of the notation used in this work; the rest of the notation is more involved and explained in the text. Let $D(h_i, A_j)$ represent the data from data structure (array) A_j that can be accessed by host h_i. We use $I(h_i, I_k, A_j)$ to denote the set of loop iterations from loop nest I_k that *can* be executed by host h_i when considering only the data structure A_j (and the HHT). In formal terms:

$$I(h_i, I_k, A_j) = \{\vec{I}: \quad \exists \vec{r} \in R(A_j, I_k) \quad \exists \vec{d} \in A_j$$
$$\text{such that} \quad [\vec{r}(\vec{I}) = \vec{d}] \quad \wedge \quad [\vec{d} \in D(h_i, A_j)] \quad \wedge \quad [\vec{I} \in I_k]\}.$$

Here, $\vec{r}$ corresponds to a reference (in I_k) and $\vec{r}(\vec{I})$ gives the array index vector accessed by iteration vector $\vec{I}$. We can write such a set for each data structure that is referenced in loop nest I_k, and then obtain:

$$I(h_i, I_k) \quad = \bigcap_{\forall j: A_j \in S(I_k)} I(h_i, I_k, A_j),$$

that is, the set of iterations from loop nest I_k that can be executed by host h_i considering all the data structures that appear in loop nest I_k. Note that, the set $I(h_i, I_k)$ just indicates the potential iterations; host h_i may or may not execute

all loop iterations in $I(h_i, I_k)$. In fact, assuming that $I'(h_i, I_k)$ is the set of loop iterations that are assigned to host h_i (after the workload balancing), we may have $I'(h_i, I_k) \subseteq I(h_i, I_k)$ or $I(h_i, I_k) \subseteq I'(h_i, I_k)$. Also, an iteration point $\vec{I}$ can belong to multiple $I(h_i, I_k)$ sets as, in determining the $I(h_i, I_k)$ sets, we consider the HHT. Specifically, if $\vec{I}$ can be accessed by h_i and if $h_i \rhd h_j$, then $\vec{I}$ will be in both $I(h_i, I_k)$ and $I(h_j, I_k)$.

5.2.3 Initial Iteration Assignment and Determining Workloads

One can employ two different strategies for the initial iteration assignment across hosts. In the first strategy, we start with the host that dominates all the other hosts in the system (e.g., h_0 in the HHT in Figure 5.3(iii)), and assign all the iterations to it (since we assume that it can execute all iteration points). Then, we iteratively distribute some of the iteration points across the other hosts in order to reach a workload distribution as balanced as possible. The second strategy adopts an opposite approach. We start by partitioning the iterations across the least powerful hosts that can execute them (i.e., across the leaves of the HHT depending on the decomposition). For example, if an iteration point can be executed by hosts h_i, h_j, or h_k and we have $h_i \rhd h_j \rhd h_k$, we give that iteration point to host h_i. After this initial assignment, we iteratively pass some of the loop iterations from these hosts to the others to balance the inter-host workload. The first strategy is called the *most powerful host policy* (MP), while the second one is referred to as the *least powerful host policy* (LP). In this work, we focus only on LP, and postpone the treatment of MP to a further study.

For an iteration point $\vec{I} \in I_k$, the least powerful host that can execute $\vec{I}$, denoted $h_{min}(\vec{I}, I_k)$ is determined by the formula,

$$\begin{array}{lllll} h_{min}(\vec{I}, I_k) & = & h_i & \text{iff} & \vec{I} \in I(h_i, I_k) \\ \wedge \quad [\neg\exists h_j & \text{such that} & h_j \rhd h_i & \wedge & \vec{I} \in I(h_j, I_k)]. \end{array}$$

Based on this, we define $I_{init}(h_i, I_k)$, the set of iterations from loop nest I_k

initially assigned to host h_i, as follows:

$$I_{init}(h_i, I_k) = \{\vec{I} : h_i = h_{min}(\vec{I}, I_k)\}.$$

In the LP policy, the $I_{init}(h_i, I_k)$ sets collectively define the initial iteration assignment, over which the rest of the process (our algorithm) tries to improve, in terms of balancing the workload. However, before starting to re-distribute iteration points in I_k, we first need to check how severe the workload imbalance is. In order to do this, we need to be able count the number of elements in each $I_{init}(h_i, I_k)$ set. Our approach to this problem is based on the solution proposed by Pugh [28]. In the rest of this work, $|I_{init}(h_i, I_k)|$ is used to refer to the number of elements in set $I_{init}(h_i, I_k)$.

It must be observed, at this point, that for a leaf node h_i in the HHT, $I_{init}(h_i, I_k)$ $= I(h_i, I_k)$, that is, their initial iteration set is the same as the set of iterations that they can potentially execute. On the other hand, for a non-leaf node h_j, we may have $I_{init}(h_j, I_k) \subseteq I(h_j, I_k)$.

5.2.4 Workload Balancing Algorithm

5.2.4.1 High-Level View

This subsection provides a high-level view of the load balancing algorithm, which is explained in more detail in the following subsections. The work balancing algorithm starts with the $|I_{init}(h_i, I_k)|$ counts (as determined using the procedure explained in Section 5.2.3) and operates on the HHT under consideration. Its goal is to re-distribute the loop iterations across the hosts such that each host takes more or less the same number of iteration points.

Our algorithm operates in two steps: bottom-up and top-down. In the bottom-up step, the traversal of the HHT in question proceeds from the leaves to the root (in post-order fashion). In this traversal, each host passes some portion of its workload to its parent (in the HHT), if its initial load is larger than the targeted average (which is N_{avg}). At the end of this bottom-up step, the root holds the iterations passed to it from the rest of the tree. The top-down step, on the other

hand, consists of a sequence of top-down traversals (in pre-order fashion). At each traversal, the host re-distributes some of the iterations passed to it across the rest of the hosts to reach a balanced load assignment. Algorithm 1 gives the code for our workload balancing scheme. The rest of this section discusses the details of this code and the two functions invoked within it. In the explanation of the algorithm, we use the terms "host" and "node" interchangeably.

5.2.4.2 ReassignHHT()

The code given in Algorithm 1 calculates N_{avg}, the ideal number of iterations each host h_i should perform, as the quotient of the total number of iterations and the total number of hosts. We define $carryout(h_i, I_k)$ as the number of iterations passed by node h_i to node h_d that dominates h_i as h_i is unable to execute all of them. The BottomToTop procedure is called with the most powerful host (denoted h_{root}) and N_{avg} as arguments. This call, when it returns, causes the initial iterations to be adjusted, such that each node performs at most N_{avg} iterations, which potentially causes $carryout(h_{root}, I_k)$ to be a value greater than 0, which is an indication that not all the iterations passed to the root can be executed by it (due to load balance concerns). In an attempt to balance the workload, N_{avg} is increased by 10% of its present value (which is a parameter that can be set to different values if desired), and the procedure TopToBottom is invoked with N_{avg} and the root as arguments. The parameter by which N_{avg} is increased can change the results obtained. If the value of N_{avg} is made too high, then the algorithm will terminate quickly but *good* load balance might not be achieved, where the level of goodness is determined by the criteria mentioned in Section 5.2. If N_{avg} is too small, then the algorithm may not finish quickly but finer load balancing can be achieved. The procedure TopToBottom tries to reassign the iterations of hosts in the HHT according to the new value of N_{avg} such that $carryout(h_{root}, I_k)$ becomes zero. Once the while loop condition in statement 3 in Algorithm 1 fails, the HHT is considered to be load balanced, and both TopToBottom and ReassignHHT return.

Algorithm 8 *ReassignHHT*()

1: N_{avg} :=$Totalnumberofiterations \div Totalnumberofhosts$
2: BottomToTop(h_{root}, N_{avg})
3: **while** carryout(h_{root}, I_k) > 0 **do**
4: $N_{avg} := N_{avg} + N_{avg} * 0.1$
5: TopToBottom($h_{root}, N_{avg}, 0$)
6: **end while**

5.2.4.3 BottomToTop()

Algorithm 2 gives the recursive BottomToTop procedure which works with the initial assignment of iterations, $|I_{init}(h_i, I_k)|$. The goal of the procedure is to assign to each host the maximum number of loop iterations that it can execute without exceeding N_{avg}. The iterations in excess of N_{avg} are passed on to the host h_d that dominates host h_i. In a recursive post-order fashion, the procedure proceeds to the deepest node, which will initially be a leaf. $carryin(h_i, I_k)$ is defined as the sum of the iterations passed to h_i by all the hosts dominated by h_i. Intuitively, $carryin(h_{leaf}, I_k)$ should be zero. After calculating $carryin(h_i, I_k)$, if the initial number of iterations assigned to host h_i is greater than N_{avg}, this means that h_i cannot accept iterations from any other node. Consequently, h_i is assigned N_{avg} iterations, and the $carryout(h_i, I_k)$ is initialized to the sum of the remaining iterations and $carryin(h_i, I_k)$, i.e., these iterations are passed to the host dominating h_i, and the procedure returns.

On the other hand, if h_i can absorb the loop iterations passed to it from the nodes it dominates, then $|I'(h_i, I_k)|$ is given the value of $|I_{init}(h_i, I_k)|$. Note that in the context of this work, an assignment to $|I'(h_i, I_k)|$, means the iterations are being added to or removed from the set $I'(h_i, I_k)$. If h_i is a leaf, $carryout(h_i, I_k)$ is set to zero as $N_{avg} > |I_{init}(h_i, I_k)|$. If h_i is not a leaf, then it might have to potentially absorb iterations from nodes that it dominates. The variable Temp in Algorithm 2 is assigned the difference between N_{avg} and $|I_{init}(h_i, I_k)|$. This value of Temp represents the number of iterations that h_i can absorb from the nodes it dominates. Next, all the children are examined in turn. For a child node, h_{child}, of h_i, if $carryout(h_{child}) > 0$, then loop iterations of h_{child} need to be absorbed. Next, it is checked to see whether Temp $\geq$ carryout(h_{child}, I_k), which means that h_i can completely absorb the iterations passed to it by h_{child}. Subsequently, Temp

Algorithm 9 *BottomToTop*(h_i, N_{avg})

```
1: for all h_i in HHT visited in post-order fashion do
2:    calculate carryin(h_i, I_k)
3:    if |I_init(h_i, I_k)| > N_avg then
4:       |I'(h_i, I_k)| := N_avg
5:       carryout(h_i, I_k) := |I_init(h_i, I_k)|N_avg + carryin(h_i, I_k)
6:    else
7:       |I'(h_i, I_k)|:= |I_init(h_i, I_k)|
8:       if h_i is a leaf then
9:          carryout(h_i, I_k):= 0
10:      else
11:         Temp:=N_avg - |I'(h_i, I_k)|
12:         for all h_child such that h_child ▷ h_i do
13:            if carryout(h_child, I_k) > 0 then
14:               if Temp ≥ carryout(h_child, I_k) then
15:                  Temp := Temp-carryout(h_child, I_k)
16:                  |I'(h_i, I_k)| := |I'(h_i, I_k)| + carryout(h_child, I_k)
17:                  carryout(h_child, I_k) := 0
18:               else
19:                  |I'(h_i, I_k)| := |I'(h_i, I_k)| + Temp
20:                  carryout(h_child, I_k) := carryout(h_j, I_k) - Temp
21:                  Temp := 0
22:               end if
23:            end if
24:         end for
25:         calculate carryin(h_i, I_k)
26:         calculate carryout(h_i, I_k)
27:      end if
28:   end if
29: end for
```

is reduced by $carryout(h_{child}, I_k)$, and following this, $|I'(h_i, I_k)|$ is increased by $carryout(h_{child}, I_k)$. Finally, $carryout(h_{child}, I_k)$ is set to zero. If, however, Temp $< carryout(h_{child}, I_k)$, then $carryout(h_{child}, I_k)$ is reduced by Temp, $|I'(h_i, I_k)|$ is increased by Temp, and subsequently, Temp is set to zero. $carryin(h_i, I_k)$ is re-calculated to reflect the changes in the "*carryout*" value of hosts that h_i dominates, following which the value of $carryout(h_i, I_k)$ is readjusted. The procedure then returns.

5.2.4.4 TopToBottom()

Algorithm 3 gives pseudo-code for TopToBottom. The procedure receives as input, from ReassignHHT, the new value of N_{avg}, the HHT via h_{root}, and *carry_reduce* which is the number of iterations host h_d, that dominates h_i, can absorb from $carryout(h_i, I_k)$. Note that the procedure ReassignHHT uses *zero* as the value of *carry_reduce* while calling TopToBottom as the root node, h_{root}, as there are no further nodes that dominate h_{root}. The HHT is traversed in pre-order fashion. For the current node being visited (h_i), the value of $carryout(h_i, I_k)$ is checked to see whether it is zero. If it is, then the sub-HHT rooted at h_i is considered to have the ideal number of iterations assigned to all its nodes, i.e., it is not possible to further balance load distribution for the nodes in the sub-HHT rooted at h_i. If, however, $carryout(h_i, I_k)$ is greater than zero, then the sub-HHT rooted at host h_i is considered to have iteration assignments that are not ideal. As a result, *tot_minus_act* is calculated to be the difference between $|I_{init}(h_i, I_k)|$ and $|I'(h_i, I_k)|$. This value is used later to determine whether h_i can absorb loop iterations from the hosts that it dominates. If $carryout(h_i, I_k) < carry\_reduce$, this indicates that host h_d that dominates h_i can completely absorb carryout(h_i, I_k), in which case, $carryout(h_i, I_k)$ is set to 0 so that this host is considered balanced in future passes of the procedure. This is different from the other possible case discussed below.

If $carryout(h_i, i_k) \geq carry\_reduce$, this means that host h_d cannot completely absorb $carryout(h_i, I_k)$. In this case, the value of *carry_reduce* is set to zero to indicate that it is exhausted. It is important to note that $carryout(h_i, I_k)$ remains unaffected since these iterations are being executed by h_d and not by h_i. If the value of $carryout(h_i, I_k)$ is reduced, this means that h_i has been assigned more iterations, not that the extra iterations are performed by h_d. Contrast this with the case when $carryout(h_i, I_k) < carry\_reduce$, where we need to indicate somehow to h_d that $carryout(h_i, I_k)$ has been accounted for and that, h_d, in future passes of the procedure, does not need to take into account host h_i while absorbing iterations from the hosts it dominates.

If the difference between N_{avg} and $|I'(h_i, I_k)|$ is greater than $carryout(h_i, I_k)$, this indicates that the increased number of iterations available to host h_i is sufficient to let h_i execute all the iterations that it is expected to execute, and thus,

$carryout(h_i, I_k)$ should be reduced to zero. This is done by increasing $|I'(h_i, I_k)|$ by $carryout(h_i, I_k)$ and reducing $carryout(h_i, I_k)$ to 0. This reduction is then recursively propagated to h_d since $carryout(h_d, I_k)$ could be affected by the reduction in $carryout(h_i, I_k)$. $carryout\ (h_d, I_k)$ is set to the difference between $carryout(h_d, I_k)$ and the reduction in the value of $carryout(h_i, I_k)$, if it is greater than zero, or to zero otherwise. Following this, h_d propagates the reduction, to the host dominating it, until h_{root} is reached. If, however, the difference between N_{avg} and $|I'(h_i, I_k)|$ is not greater than $carry\_reduce$, it is an indication that the increased iterations reduce the value of $carryout(h_i, I_k)$ partially. Consequently, $carryout(h_i, I_k)$ is recalculated as the difference between $|I'(h_i, I_k)|$ and N_{avg}. As in the earlier case, this reduction is propagated recursively to host h_d that dominates h_i. The value of tot_red is calculated as N_{avg} -$|I'(h_i, I_k)|$. If $tot\_minus\_act < 0$ and $|I_{init}(h_I, I_k)| > N_{avg}$, then tot_red is reduced by the value tot_minus_act. If $tot\_red < 0$, then h_i cannot absorb iterations form its children, and therefore, tot_red is set to 0. Finally, TopToBottom is called recursively for all the nodes dominated by h_i with the $carry\_reduce$ argument being a proportional portion of tot_red that depends on the values of $carryin(h_i, I_k)$ and carryout(h_j, I_k) of all h_j that are immediate children of h_i in the HHT.

5.2.5 Example

In this subsection, we explain the working of our load balancing algorithm through the example code, based on the Gauss Seidel method [61], shown below written in a pseudo language. The HHT considered for this example is shown in Figure 5.4(a). The data space decompositions for arrays A and B given in Figure 5.4(b), and Figure 5.4(c) shows the initial workloads for the hosts. Figure 5.4(d) shows the snapshot of the HHT after the BottomToTop procedure operates. As the tree is unbalanced, procedure TopToBottom is called. Figure 5.4(e) shows the snapshot of the HHT after the first pass of TopToBottom. As the first pass of TopToBottom does not enough to reduce the complete load imbalance in this example, TopToBottom is run repeatedly until the HHT is balanced. The results of further passes of the procedure are shown in Figures 5.4(f), (g), and (h).

for$(i = 2 \quad to \quad N - 1)$

Algorithm 10 $TopToBottom(h_i, N_{avg}, carry\_reduce)$

```
1: for all hosts in the HHT in pre-order fashion do
2:   if carryout(h_i, I_k)=0 then
3:     Return
4:   end if
5:   tot_minus_act := |I_init(h_i, I_k)|-|I'(h_i, I_k)|
6:   if carryout (h_i, I_k) - carry_reduce ≤ 0 then
7:     carryout(h_i, I_k) := 0
8:     return
9:   else
10:    carry_reduce := 0
11:    if N_avg-|I'(h_i, I_k)| > carryout(h_i, I_k) then
12:      |I'(h_i, I_k)| := |I'(h_i, I_k)|+carryout(h_i, I_k)
13:      carryout(h_i, I_k) := 0
14:      return
15:    else
16:      carryout(h_i, I_k) := N_avg-|I'(h_i, I_k)|
17:      adjust reduced carryout recursively to h_j s.t. h_i ▷ h_j
18:      tot_red:= N_avg-|I'(h_i, I_k)|
19:      if ((tot_minus_act > 0) && (|I_init(h_i, I_k)| > N_avg)) then
20:        tot_red := tot_red-tot_minus_act
21:      end if
22:      if tot_red < 0 then
23:        tot_red := 0
24:      end if
25:      call TopToBottom recursively for all h_j s.t. h_j ▷ h_i and absorb tot_red
         number of iterations proportionally from them
26:    end if
27:  end if
28: end for
```

```
for(j = 2   to   N − 1)
  B[i, j] := (A[i − 1, j]+ A[i + 1, j] + A[i, j − 1]+ A[i, j + 1]) * 1/α ;
endfor
endfor
```

5.2.6 Locality Concerns

While balancing the workload across the hosts is important, this itself does not guarantee fast execution. This is because even if the load is perfectly balanced (in

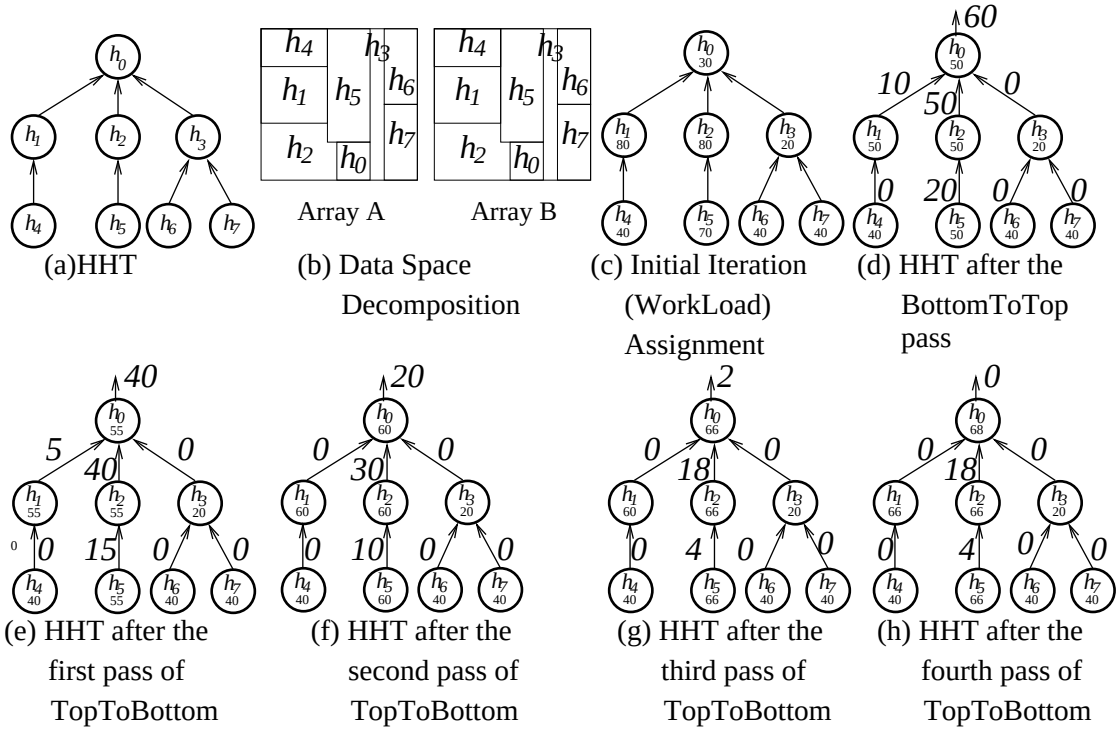


Figure 5.4. (a) An example HHT. (b) Example data decompositions. (c) Initial workload assignment. (d) Situation after BottomToTop. (e-h) Situation after different passes of TopToBottom. The numbers within the nodes denote the current loads (in terms of loop iterations) and the numbers along the arrows indicate carryouts.

terms of loop iterations): different iterations can have different execution times (cycles). This can occur due to at least two reasons: the conditional control flow within the loop body and cache memory behavior. The first of these means that if there is an IF statement within the loop body, different loop iterations can take different branches of this statement, and this can lead to different loop iterations taking different amounts of time to finish the loop body as different branches can take different execution times. Branch prediction could be used to make control flow decisions based on which the computation times of different iterations could be determined. We currently do not address this problem as our experience with different embedded applications show that this case is very uncommon. The second reason, however, is more likely to occur, and originates from the possibility that different loop iterations access different data, and depending on the current locations of these data (e.g., cache versus main memory), data access time (hence, loop body execution time) can be different. Therefore, in our context, from the

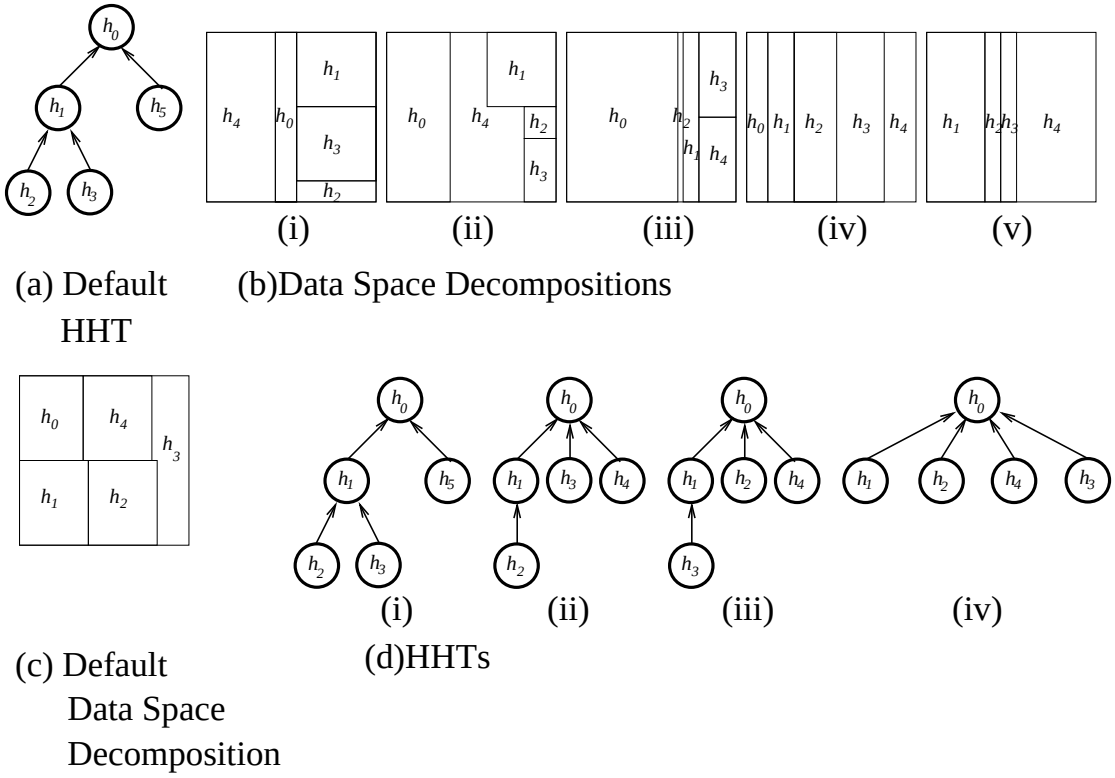


Figure 5.5. Setup for the first experimental scenario: (a) and (d) HHTs. (b) and (c) data decompositions.

perspective of a particular host, it is most beneficial if this host accesses the data with reuse, i.e., the loop iterations that are assigned to it (after our workload balancing algorithm) use the same set of data as much as possible. One place in our algorithm that this can be taken care of is when we a host passes some iteration points to another host to balance the workload. By being selective about which iterations to pass, we can achieve better execution times.

Let $E(h_i, I_k)$ denote the current set of iterations from loop nest I_k that are assigned to host h_j. The set of data items from array A_j that are accessed by the iterations in $E(h_i, I_k)$ can be expressed as:

$$\begin{aligned} G(h_i, I_k, A_j) \;&=\; \{\vec{d}: \quad \exists \vec{r} \in R(A_j, I_k) \\ &\text{such that} \qquad [\vec{r}(\vec{I}) = \vec{d}] \quad \wedge \quad [\vec{I} \in E(h_i, I_k)]\}. \end{aligned}$$

As before, $\vec{r}$ corresponds to a reference and $\vec{r}(\vec{I})$ gives the array index vector accessed by iteration vector $\vec{I}$. Let us now assume that host h_l wants to pass a subset of its current set of iterations, $E(h_l, I_k)$, to host h_i. Assume further that the size of this subset needs to be R, where $R \leq |E(h_l, I_k)|$. The idea is to select the most beneficial R iterations such that data locality is improved. Let us use $K(h_l, h_i, I_k)$ denote the subset of iterations that will be passed from h_l to h_i. The data elements from array A_j that will be accessed by the iteration points in $K(h_l, h_i, I_k)$ can be expressed as:

$$\begin{aligned} H(h_l, h_i, I_k, A_j) \;=\; & \{\vec{d}: \quad \exists \vec{r} \in R(A_j, I_k) \\ \text{such that} \qquad & [\vec{r}(\vec{I}) = \vec{d}] \quad \wedge \quad [\vec{I} \in K(h_l, h_i, I_k)]\}. \end{aligned}$$

For temporal data locality, sets $H(h_l, h_i, I_k, A_j)$ and $G(h_i, I_k, A_j)$ should have common elements, and for spatial data locality they should access the elements that reside on the same cache line. To be general, let $\Pi[h_l, H(h_l, h_i, I_k, A_j), G(h_i, I_k, A_j)]$ give the estimated number of misses (due to accesses to array A_j) that would be experienced by host h_l when the iteration points in question are passed from host h_l to host h_i. Similarly, let us use the term $\Pi[h_i, H(h_l, h_i, I_k, A_j),\ G(h_i, I_k, A_j)]$ to denote the number of estimated misses that would be experienced by host h_i as a result of this iteration transfer. The goal must be to select the $K(h_l, h_i, I_k)$ set in such a way that $\Pi[h_i, H(h_l, h_i, I_k, A_j), G(h_i, I_k, A_j)] + \Pi[h_l, H(h_l, h_i, I_k, A_j), G(h_i, I_k, A_j)]$ is minimized. To achieve this, our approach works as follows. We first identify the iterations in $E(h_l, I_k)$ that do not exhibit any temporal reuse in h_l but would lead to temporal reuse if they are executed by h_i. Let R' denote the number of such iteration points. If $R' \geq R$, then our job is easy as we can pass R of these R' iterations from h_l to h_i. Otherwise, we identify the iterations in $E(h_i, I_k)$ that do not exhibit any spatial reuse in h_l but would lead to spatial reuse if they are executed by h_i. Assuming that there are R'' such iterations, if $R' + R'' \geq R$, we are done. If not, then we select $R - (R' + R'')$ more iterations from $E(h_l, I_k)$ – even if they would not lead any improved locality in h_i – and move them from h_l to h_i.

Note that our approach minimizes the execution time of a given application

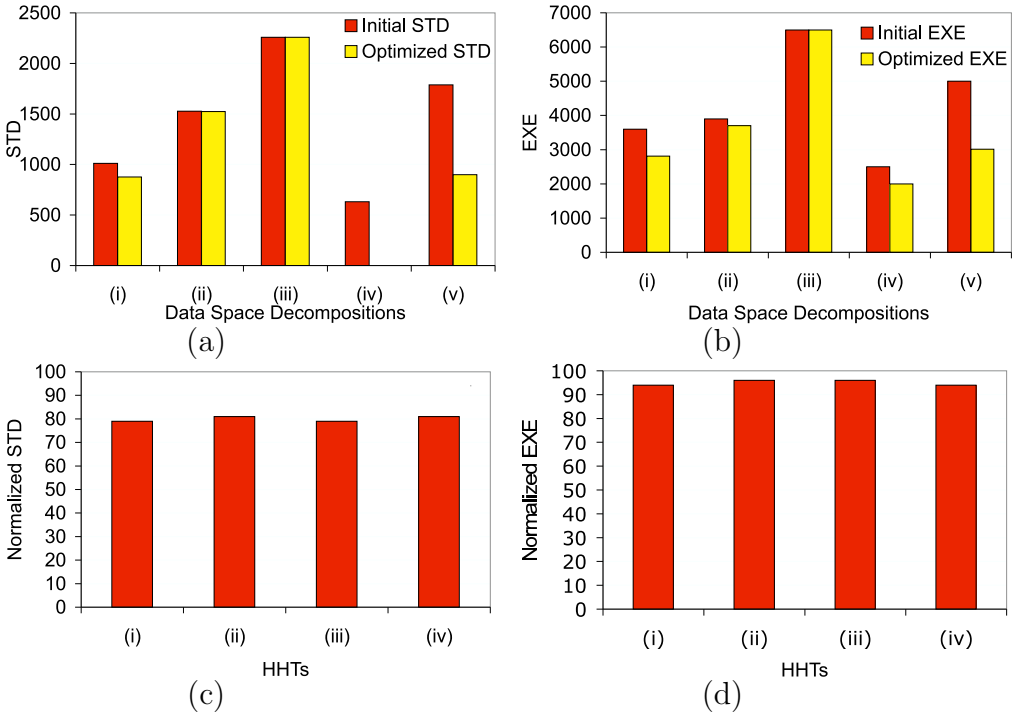


Figure 5.6. STD and EXE results.

under the constraint that no security leak is allowed which in our context means that access control is violated. If the resulting execution time exceed the specified execution time bound, this means that our approach could not find a solution under this specified bound. One could also study how much security leak needs to be allowed to satisfy the performance bounds in such situations. However, since we assume that our application domain cannot tolerate any security leaks, we do not discuss this issue further here.

5.3 Experimental Evaluation

In this section, we present an experimental evaluation of our workload balancing algorithm. We focus on two example scenarios. The first scenario deals with embedded secure image processing at real time, where a number of hosts collectively perform a smoothening operation on an image, but the different parts of the image can be manipulated by different hosts. The second scenario focuses on an encryption application. Each host interprets a portion of a given data file

using a different key (that is known only to that host). For both the scenarios, we conduct two types of experiments. First, we fix the HHT and change the data decomposition, and then, we fix the data decomposition and the number of hosts, and change the structure of the HHT. We use two metrics to quantify the quality of the workload partitioning determined by our approach: (a) standard deviation between the workloads of the hosts and (b) the largest number of iterations assigned to a host. In the rest of this section, these two metrics are denoted as STD and EXE, respectively. Note that, as the load balancing step is performed once at compile time, it does add to the runtime of the application.

Figure 5.5(a) shows the default HHT used for the secure image scenario. In this scenario, each host processes a portion of an image. The operation performed is smoothening, i.e., each pixel (represented by an array element) is updated using the values of its four neighbors. The graphs in Figures 5.6(a) and (b) give, respectively, the STD and EXE metrics for this scenario under the data decompositions shown in Figure 5.5(b). For each data decomposition, we have two bars: one for the original (initial) workload and the other for the optimized workload (using our approach). Our approach improves both the metrics for all the data decompositions experimented. To be precise, our approach improves STD and EXE by about 33% and 19%, respectively, on the average. In the next set of experiments, we use the data decomposition in Figure 5.5(c) and use different HHTs shown in Figure 5.5(d). The experimental results are given in Figures 5.6(c) and (d) for STD and EXE, respectively. Since the EXE and STD values of the initial distribution (iteration assignment) do not depend on the structure of the HHT, each optimized result is given as a fraction of the original value. As in the previous set of experiments, we observe that our algorithm improves workload distribution. Specifically, it improves STD and EXE by approximately 20% and 5%, respectively, on the average.

The data encryption scenario consists of two data structures. The first one is a two-dimensional array that holds the data to be protected. The second one is a key structure, and each host can access its own key. The first data structure can be accessed in a fashion dictated by the data decomposition. Our approach improves STD and EXE by around 33% and 16%, respectively, when averaged over different data decompositions tried. Further, when averaged over all HHTs used,

our approach improves STD and EXE by 30% and 29%, respectively.

5.4 Related Work

Secure code partitioning has been studied in the context of information flow theory. Static methods as well as dynamic techniques exist to study the flow of secure information through a system. The initial work in using program analysis to prove that an information flow is secure was [62]. Jif [63] and CQUAL [64] are examples of current tools that perform static analysis of a program through analysis of the security types present in the program. SELinux [65] is an operating system that provides support for security aware applications. Other distributed operating systems that have been proposed such as [66] which execute code across multiple hosts keeping performance in mind. Load balancing across different hosts in a distributed environments is a well researched topic [61]. In contrast to the distributed environment, our environment does not consider communication costs. [67] proposes a framework, using which software modules can be divided into open and hidden components, which are then executed on insecure and secure machines, respectively. Prior compiler work on secure computation includes [68]. Maybe the most relevant prior work to ours is [52, 54], where the authors present secure program partitioning to protect confidentiality of data. They are primarily interested in enforcing confidentiality policies at the language level. In contrast, our goal is to automate secure code partitioning within an optimizing compiler. In addition, we want to minimize execution time of the parallelized application without compromising sensitive data.

5.5 Concluding Remarks

Widespread use of parallel processing in the embedded computing world and increase in data/code sharing bring an important problem: ensuring proper communication between hosts that operate on secure data. While making sure that each data structure is manipulated only by authorized hosts is a must, one also wants to reduce parallel execution time as much as possible. This work presents a workload distribution algorithm that partitions an embedded code between different

hosts such that data security is not compromised and execution time is reduced as much as possible. Our approach takes (as input) the application code to be partitioned, a host hierarchy, and data decomposition across the hosts, and generates (as output) the code partitions, each of which is assigned to a host. We tested our approach using two example scenarios, namely, secure image processing and encryption, and found that it improves workload balance significantly across different data decompositions and host hierarchy trees.

Chapter 6

A Reuse Oriented Approach to Manage Encrypted Data in SPM Based Systems

6.1 Introduction

Many of today's embedded computing systems typically hold and manipulate sensitive data such as personal identification and financial information. Successful attacks on such systems have led to the loss of large amounts of data and the results of confidential information including medical data [69, 70, 71]. However, most of these systems also operate under tight performance requirements. Therefore, balancing security and performance requirements is critical.

Data confidentiality can be defined as ensuring that information is accessible only to those authorized to have access and is one of the cornerstones of any computing system where security is a concern. Encryption has long been employed as one of the primary mechanisms to ensure confidentiality, as is evident from many prior studies [72, 73, 74]. One of the important problems associated with encryption however is the performance degradation it can bring, especially when decryptions sit on the critical path of execution.

Our goal in this work is to explore the performance/security tradeoffs in the execution of applications on SPM (scratch-pad memory) based systems. SPMs

are frequently employed by modern systems due to their fast access latencies, low power consumption, and good performance predictability (as far as worst case execution times are concerned). In our execution scenario, off-chip main memory is assumed to lie outside the trust base, and consequently, data should be stored in the main memory in an encrypted form as much as possible. Normally, decryptions take place when data is brought from the off-chip memory to the SPM, and data is encrypted back when it is stored in the off-chip memory. However, during the execution of a data-intensive application, the rate of data traffic between the SPM and off-chip memory can be very high. As a result, we may incur frequent encryptions/decryptions, which may cost significant execution cycles and power consumption, and this may not be tolerable in many embedded execution environments.

Clearly, when the cost of ensuring data confidentiality is too high, an alternate option is not to employ encryption at all. While this is certainly a performance-efficient solution, it may not be acceptable to a system with security concerns. This work explores a *middle solution* between these two extremes, as shown in Figure 6.1. More specifically, it studies the problem of how much performance can be saved when we are allowed to sacrifice some confidentiality during the course of execution. While one can develop different schemes that can be used for exploring this tradeoff, in this work we study a *data reuse based* approach. The intuition behind this approach is that data that is written out of the SPM and read back at a later time may be left in an unencrypted form in memory as long as certain conditions are satisfied. We can summarize our main contributions as follows:

• We present a quantitative analysis of the performance gain and the confidentiality loss when only select data blocks are encrypted when data is written back to the off-chip memory.

• We propose a metric of evaluation which captures both performance gain and confidentiality loss, and can be used for studying this tradeoff.

• We present a data reuse oriented approach to encryption. This approach (which employs an oracle to see future data accesses) uses encryption during memory writes, only if the reuse distance of the data involved is larger than a threshold value.

• We propose a profile based scheme, that mimics the behavior of the oracle

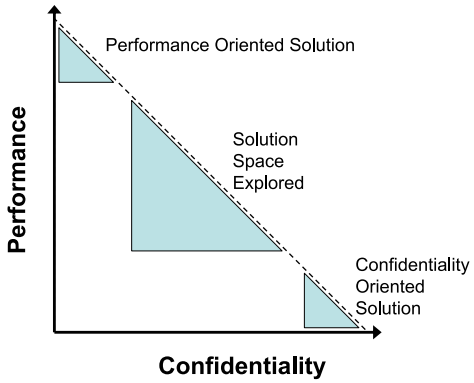


Figure 6.1. The solution space we explore offers a compromise between the pure performance oriented and the pure data confidentiality oriented solutions. A performance oriented solution offers very low data confidentiality. Similarly, a data confidentiality oriented solution would suffer in terms of performance. Our approach allows an application designer to explore the solution space between these two extremes and choose a solution that is acceptable to the design at hand.

based approach to data encryption and leverages the knowledge gained from our quantitative analysis.

Based on our results, we conclude that proper software support that can be tuned by the programmer based on the desired level of confidentiality and acceptable level of performance loss is very important. Targeting SPM based platforms, to our knowledge, this is the first effort that studies this tradeoff, and reports empirical data.

The importance of this work is that it allows an application designer to tune her application according to performance and confidentiality requirements. Moreover, if these requirements change, it is very simple to tune the application differently. The motivation for employing a software/compiler based approach to this problem is that the information about the reuse of data blocks can only be gathered by looking at the entire window of execution of the application. This is not possible through hardware methods that can only look at a limited window of memory operations. Intuitively, the greater the reuse information that is available, the more accurate the tunability of the application by the application designer.

Section 6.2 presents the high level view of our target system, and outlines the analysis performed in this work. Section 6.3 gives a brief introduction to SPM and our data replacement policy. Section 6.4 explains the encryption and decryption

operations that are involved in transferring data blocks between the SPM and off-chip memory. Section 6.5 introduces the metrics used to quantitatively evaluate the performance and data exposure of two extreme security schemes. Section 6.6 discusses a reuse based approach to securing data blocks, and Section 6.7 presents a metric that combines performance and data exposure to evaluate the behavior of the reuse based approach. Section 6.8 presents a profile guided approach to mimic the reuse based approach. Section 6.9 analyzes the sensitivity of the approach. Section 6.10 discusses the related work and the chapter is concluded in Section 6.11.

6.2 High Level View

This section presents a high level view of a secure SPM based system that we focus on in this work and also outlines the analysis performed. The system we target can be divided into a *trusted portion* and an *untrusted portion*. The trusted portion consists of the CPU with a local store of data such as registers, cache or SPM. The untrusted portion on the other hand consists of the off-chip memory which may be accessed by other untrusted processes. Further, malicious parties may observe the transactions between the processor and the off-chip memory. In order to thwart any malicious attack from resulting in a loss of data confidentiality, data is preferred to be in an encrypted form while it resides in the untrusted portion of the system. In the trusted portion, data exists in an unencrypted form so that it may be operated on by the CPU. When data is transferred from the untrusted portion to the trusted portion, it is decrypted in between. When it is transferred from the trusted portion to the untrusted portion, it is encrypted. Decryptions and encryptions are performed by a *cryptographic engine* (or *crypto-engine* for short) that acts as a gateway for the movement of data between the trusted portion and the untrusted portion of the system (see Figure 6.2). The scratch-pad memory itself can also be termed as a software controlled cache because special instructions in the application code specify what data must be brought into the SPM and what data must be removed from the SPM. These instructions are placed into the code by the application writer manually or by an optimizing compiler automatically in a security agnostic manner, and are based solely on the data requirements of an

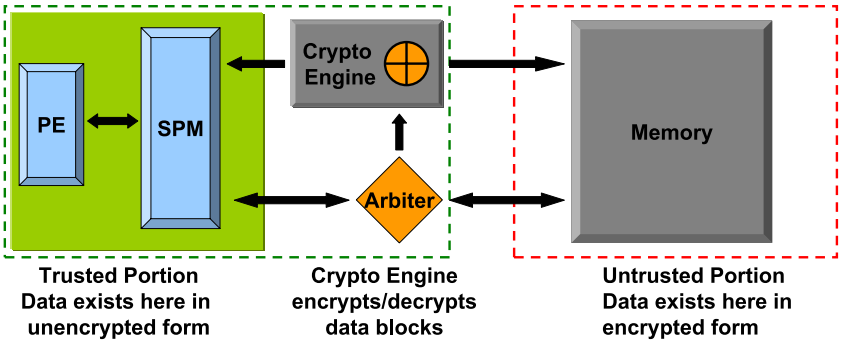


Figure 6.2. The high level view of our SPM-based target system. This system can be used to improve the performance of secure applications. Data exists in the off-chip memory in encrypted form and in unencrypted form in the on-chip SPM.

application. The instructions are tuned to bring the appropriate data into the SPM during the course of execution of the application. In this work, the order in which the data is brought in and removed from the SPM is called the *memory trace* of the application.

In this work, the memory trace of applications running on an SPM based system is generated and different security scenarios are studied using the following steps:

1. Applications are instrumented with instructions that simulate the transfer of data between the memory and SPM.

2. A trace of the behavior is generated by running the application and storing the addresses of the data being transferred.

3. The trace is then studied to generate metrics on the performance of the application and the exposure of data based on the time taken to encrypt and decrypt data blocks transferred between memory and the time data spends in unencrypted form in memory. This analysis is first conducted for two extreme scenarios; one in which the data is always kept in encrypted form in memory and another, where data is kept in decrypted form as much as is possible. The first scenario helps maximize confidentiality while the second maximizes performance.

4. Using an oracle based system, the confidentiality requirement that all data that is in the untrusted portion should be encrypted is relaxed in order to

study the dependence between exposing data and the memory performance of an application. Specifically, data that is written out of the SPM once and known to be read back soon, is kept in unencrypted form in memory. This future knowledge of instructions is obtained by examining the memory trace that is generated for an application. Therefore, the oracle system is ideal in the sense that decisions are made with complete knowledge of data reuse patterns. A profile based approach that does not depend on the knowledge on future instructions is also developed. The profile based scheme results in am implementation that mimics the performance and exposure behavior of the reuse based scheme. It is important to note that the applications that we are interested in consist of a control thread, and a data-processing thread. We sidestep the problem of revealing highly sensitive information such as passwords, by only exposing the data accessed by the data-processing thread. In the rest of the chapter, the term application refers to the data-processing thread of the application.

6.3 SPM Management Policy

This section briefly discusses how the applications were instrumented, and explains the SPM management policy employed in our work. The management policy used in this work follows the principle that all the required data is fetched from off-chip memory into the SPM before the data is used by the CPU. Applications tested in this study are codes whose primary data structures (arrays) are accessed within nested loops. Note that such applications frequently appear in data-intensive embedded computing (e.g., video stream processing). Each benchmark was analyzed and instructions to transfer data blocks between the SPM and memory were inserted.

The instruction *SPM_rd_into(DS,Arr_Start,SPM_Start,Size)* is used to bring $Size$ amount of data from the data structure DS, starting at Arr_Start into the SPM location SPM_Start. The instruction placement was performed after careful analysis of the applications.

Arrays accessed in the application are categorized according to their size, the type of operation (read/write) performed on them, and the duration of time that

```
void sum_rows(int A[N], int B[N][N]){
  for(i=0; i < N; i++){
    for(j=0; j < N; j++){
      A[i] += B[i][j];
      B[i][j]++;
    }
  }
}
```

(a)

```
void SPM_rd_into(char Array_Name, int Arr_Start, int SPM_start, int Size);
void SPM_wr_back(char Array_Name, int Arr_Start, int SPM_start, int Size);

void sum_rows(int A[N], int B[N][N]){
  SPM_rd_into(A, 0, 0, 1);
  SPM_rd_into(B, 0, 1, (int) N/2);
  replacement_index=0;
  for(i=0; i < N; i++){
    for(j=0; j < N; j++){
      A[i] += B[i][j];
      B[i][j]++;
    }
    if(i % (int)(N/4)==0 && i>0){
      SPM_location = 1 + replacement_index * (int) N/4;
      replacement_index =  (replacement_index + 1) % 2;
      SPM_wr_back(B, i-(int)N/4, SPM_location, (int) N/4);
      if(i < 3* (int) N/4)
        SPM_rd_into(B, i+(int) N/4, SPM_location, (int) N/4);
    }
  }
  replacement_index =  (replacement_index + 1) % 2;
  SPM_wr_back(B, i-N/4, SPM_location, (int)N/4);
  SPM_wr_back(A, 0, 0, 1);
}
```

(b)

```
RD: Array A locn. 0  to     SPM locn. 0, size 1*16
RD: Array B locn. 0  to     SPM locn. 1, size 8*16
WR: Array B locn. 0  from   SPM locn. 1, size 4*16
RD: Array B locn. 8  to     SPM locn. 1, size 4*16
WR: Array B locn. 4  from   SPM locn. 5, size 4*16
RD: Array B locn. 12 to     SPM locn. 5, size 4*16
WR: Array B locn. 8  from   SPM locn. 1, size 4*16
WR: Array B locn. 12 from   SPM locn. 1, size 4*16
WR: Array A locn. 0  from   SPM locn. 0, size 1*16
```

(c)

Figure 6.3. (a) Original code. (b) Instrumented code with SPM instructions. Each instruction directs a certain number of rows to be written out of the SPM or read into the SPM. (c) Trace of instructions formed by this code when N = 16.

they are needed in the SPM. If a data block is not modified during its stay in the SPM, it is only read into the SPM and not written out. Other blocks that are modified while in the SPM must be read first into the SPM and written out of the SPM into the off-chip memory when required. The instruction *SPM_wr_back* is used to simulate the writing back of data from the SPM to the off-chip memory.

In this SPM management policy, when certain data blocks are required continuously for an extended period of the execution of the application, they are retained

in the SPM throughout that period. On the other hand, some arrays are accessed in a regular pattern which involves certain parts of the array being required in the SPM at any given time. To simulate this; periodically, data blocks that are required for execution are read into the SPM and blocks currently in the SPM are written out first if required. The space allocated to such arrays is divided into two halves. At any point, the CPU performs computation on one half of the array data blocks, while the other half is written back if required and new data blocks are read into its place. As an example, Figure 6.3(a) shows a sample code fragment and Figure 6.3(b) shows the corresponding instrumented code. Array A is a one-dimensional array that is needed throughout the execution, and array B is a two dimensional array, that is needed row by row for execution. Since array A is needed throughout the execution, it is read into the SPM at the beginning and read out at the end. Parts of array B, in groups of 4 rows each, are brought into the SPM in the order in which they are needed by the computation. Note that, array B is written out because its elements are incremented. If it was only read and not modified while in the SPM, it would not need to be written out of the SPM.

By running the application through this policy, the two functions, *SPM_rd_into* and *SPM_wr_back*, record the series of dynamic memory instructions that are issued and the data blocks that are accessed. Figure 6.3(c) shows the trace of memory operations of the instrumented code. Such a trace formed by code instrumentation is used to analyze the SPM behavior of applications using the evaluation metrics described in succeeding sections. It is important to emphasize that, while our approach uses this particular SPM management scheme, it can be used with other SPM management schemes as well (e.g., [75, 76]). That is, the choice of the underlying SPM management scheme is orthogonal to the focus of our approach.

6.4 Use of the Cryptographic Engine

The default behavior of the target system is to keep *all* the data in memory in encrypted form. Intuitively, keeping data in encrypted from would entail a decryption before its use, and writing data in encrypted form requires an encryption operation before it is written. Our work modifies this default behavior to perform *selective*

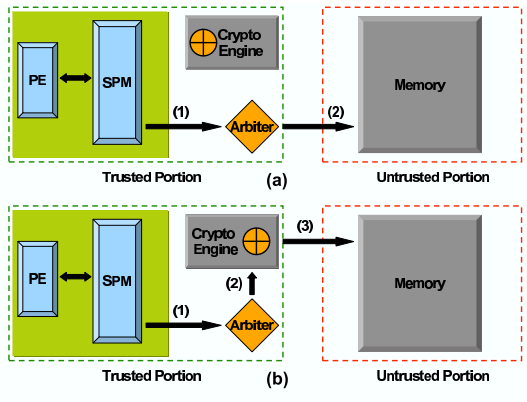


Figure 6.4. The target system and the steps involved in transferring a data block from the SPM to the off-chip memory.

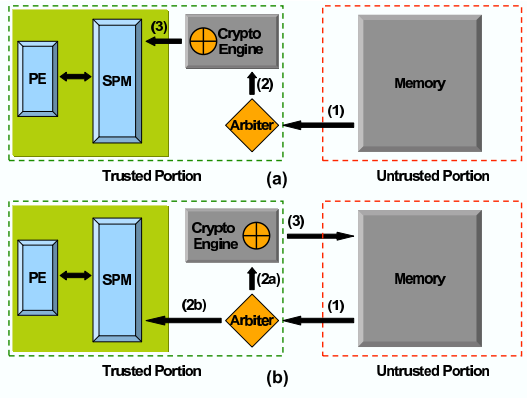


Figure 6.5. The target system and the steps involved in transferring a data block from the off-chip memory to the SPM.

encryption and decryption. The modified behavior is used to support our analysis of selectively exposing data in order to study its positive effect on performance. Figures 6.4 and 6.5 illustrate the modified operations of the cryptographic-engine that lies between the CPU and the off-chip memory.

Figure 6.4 shows the steps involved in transferring a block of data from the SPM to the off-chip memory. Figure 6.4(a) deals with the case when data is written to the off-chip memory without encryption. In step (1), the SPM transfers the data block to the arbiter along with a decision as to whether it should be encrypted (which in this case is to not encrypt). In step (2), the block is written into the memory. Figure 6.4(b) deals with the case when data is written to the memory in

an encrypted form. In step (1), the SPM transfers the data to the arbiter along with a decision to write the data into the off-chip memory in encrypted form. In step (2), the data is sent to cryptographic engine where it is encrypted. In step (3), the block is written into the memory. In our current implementation, a bit (called the *status bit*) in the top byte of the data is set to indicate whether the data block is currently in encrypted or unencrypted form. Thus, when the data block is read, we are aware of whether it is encrypted or not. Figure 6.5 shows the steps involved in transferring a block of data from the off-chip memory to the SPM. There are two cases to consider depending on whether the data block in memory is in encrypted form or in unencrypted form. The first case in which the data is in encrypted form in off-chip memory is shown in Figure 6.5(a). In step (1), the data block is read from off-chip memory, and the arbiter examines the status bit of the data and determines that it is encrypted. The block is therefore sent to the cryptographic engine for decryption (step (2)). Following decryption, the data is sent to the SPM (step (3)).

The other case, illustrated in Figure 6.5(b), occurs when the block is in decrypted form in off-chip memory. In step (1), the data block is transferred to the arbiter where it is detected that the data block is in unencrypted form. In this case, it is sent to the crypto-engine for encryption (step (2a)) and to the SPM (step (2b)) simultaneously. When the data block is in the SPM, it may be used by the CPU. When the data block reaches the cryptographic engine, the block is encrypted and written back to the address it was read from (step (3)). Note that the encryption of the plain data block is off the critical path and does not affect the time in which the block is brought into the SPM. The reason for encrypting the block is that the block was kept unencrypted to save on decryption time when it is read. Once it has been read, it brings no further benefits to keep it unencrypted.

6.5 Evaluation of Extreme Schemes

Based on the code instrumentation technique described in Section 6.3, the memory traces of four array-based/data-intensive benchmarks were created. Note that the memory traces are independent of the input data set and are dependent only on the control flow such as loops in the code. Furthermore, there were no control-flow

Benchmark	Static Instructions		Dynamic Instructions		Data Transfer
	Reads	Writes	Reads	Writes	Size (MB)
ADI [77]	15	15	576	576	1.68
SWIM [78]	61	19	3319	1642	4.84
TSF [79]	5	5	1025	1020	15.97
LU [80]	13	13	33952	39456	4.48

Figure 6.6. Details of the benchmarks studied.

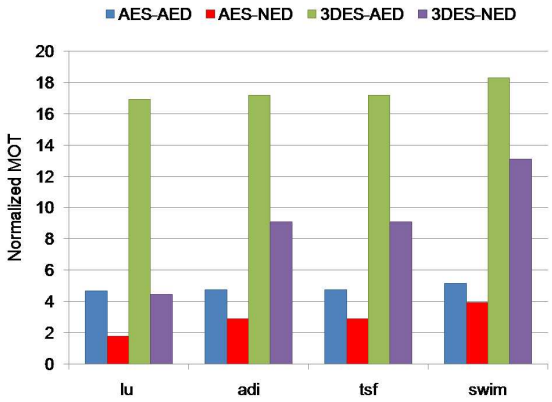


Figure 6.7. The normalized *MOT* values under the Never Encrypt Decrypt and Always Encrypt Decrypt schemes.

statements in the code that would result in different traces from what is created for one set of input data. The details of the benchmarks used in this study are given in Figure 6.6. In order to study the tradeoffs between the benefits and risks of keeping data in the off-chip memory in an unencrypted form, two metrics, the total *memory operation time* (MOT) and *exposure* (E), are introduced and measured.

•*MOT :* This represents the total time spent in reading, writing, encrypting, and decrypting the data blocks in the critical path of bringing new data blocks into the SPM.

• *E :* This captures the total amount of exposed data (over the entire execution period) as a percentage of the total data in off-chip memory. In this work, we study two versions of this metric, average exposure (AE) and maximum exposure (ME). Average exposure is defined as the average amount of data exposed considering the entire execution, and maximum exposure is defined as the maximum amount of data exposed at anytime during execution.

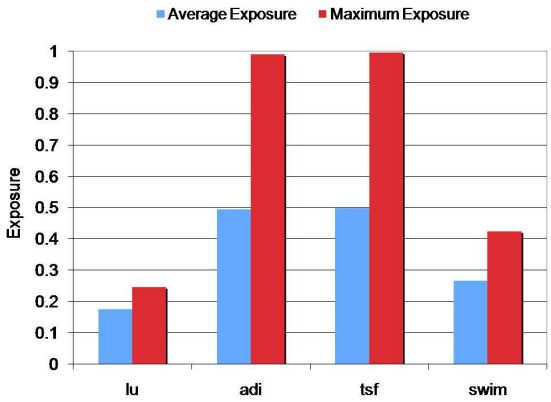


Figure 6.8. The normalized E values under the Never Encrypt Decrypt scheme.

We use these two metrics to quantify the memory behavior of two extreme schemes. The first scheme keeps all the data blocks in encrypted form while they are in the off-chip memory. As a result, data blocks must always be decrypted when they are read into the SPM and every block written back into the off-chip memory must be encrypted. This scheme entails the maximum number of encryption and decryption operations possible for a given trace and is called the $AlwaysEncryptDecrypt$ (AED) scheme.

The second scheme keeps any data that is written out of SPM in unencrypted form during the course of execution if that data block is read back into the SPM at a later point in the execution. Hence, this scheme eliminates the decryption latency in the critical path for all future reads to that block. The knowledge of whether a block will be read in the future is gained from the memory trace. This scheme is the most unsecure scheme possible and is called the $NeverEncryptDecrypt$ (NED) scheme. A point to note is that we do not keep blocks that are only read into the SPM in decrypted form in the memory. This is because, in our implementation, a block may switch from encrypted form to unencrypted form in off-chip memory only if there is an *SPM_wr_back* instruction that writes that block and if that *SPM_wr_back* instruction chooses to write data in unencrypted form.

In order to implement the NED scenario, a change to the operation of the arbiter in the target system is needed. Recall that, in our target system, reading an encrypted block from the off-chip memory would mean that the same block is encrypted by the cryptographic engine and also written back (see Figure 6.5(b)).

However, we do not want this to happen as there may be further reads of the same block later in the trace (as this is the NED scheme). Therefore, we disable the arbiter's action of simultaneously writing back a read block to the memory in unencrypted form.

As the value of MOT is determined by the number of encrypted read, encrypted write, plain read and plain write operations, the choice of the encryption scheme employed affects the value of MOT. More specifically, different encryption schemes typically have different latency requirements to decrypt and encrypt data blocks. In this study, we measured the times taken under the AES [73] and the triple DES [74] schemes, and present our MOT results using both encryption mechanisms. The implementations used for AES and DES used are from the OpenSSL cryptography toolkit [81]. The time to decrypt (encrypt) a single block is calculated by performing one million decrypt (encrypt) operations and taking the average time over all the operations.

Figure 6.7 shows the normalized MOT values across the four benchmarks for both the encryption schemes and the two extreme scenarios described above. The MOT values shown are normalized to that of the scenario in which no encryption or decryption is performed, i.e., a computing system without any security implementation. It is noticed that the MOT value is always lesser in the NED case and higher in the AED case.

Figure 6.8 shows the AE and ME values across the four benchmarks. We only study the values for the NED scheme as AED never exposes data in an unencrypted form. It should be noted that the choice of encryption scheme does not affect the exposure of the data, as E is dependent on the number of blocks that are exposed, not on latency of encryption and decryption. In the *lu* and *swim* cases a small portion of the total data blocks is exposed. In the *adi* and *tsf* cases, almost all of the data blocks in the application are exposed at some point during the execution.

The trade-offs between performance and exposure in the *NED* case becomes apparent once the results for the *lu* benchmark are examined more closely. We can see that, for less than 25% exposure of data, the fall in MOT is greater than 60%. Similar observations can be made for some other benchmarks as well. Hence, an application designer can choose between the exposure of data and performance

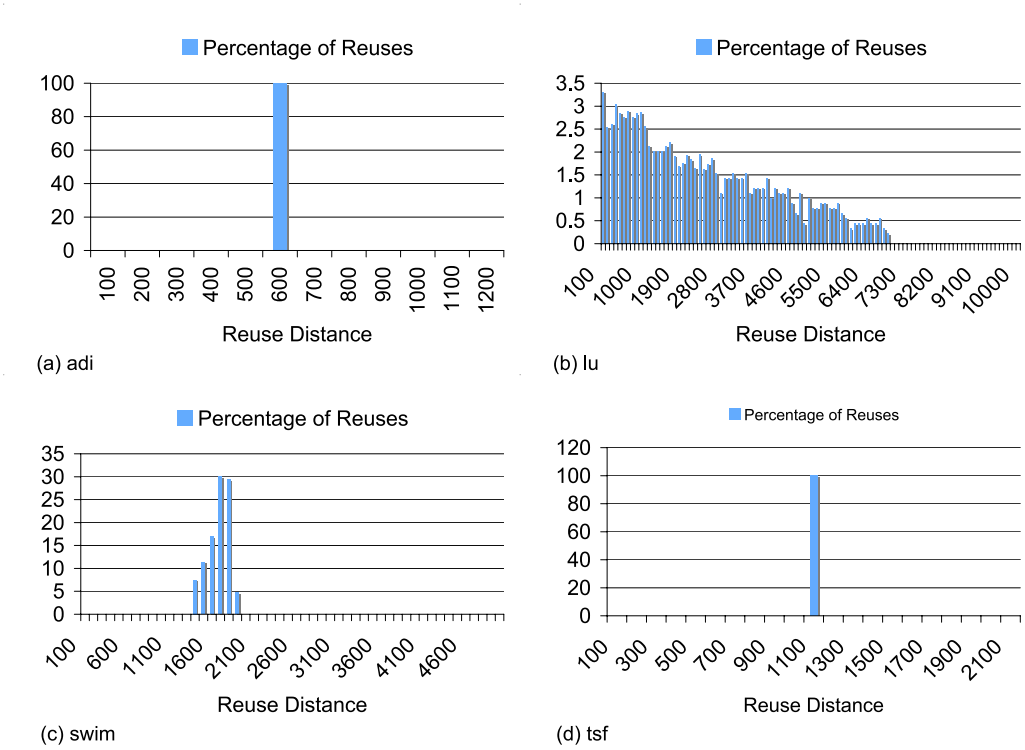


Figure 6.9. Distribution of data reuses according to reuse distance. Reuse distance is measured in terms of memory instructions.

gain. This analysis therefore allows us to study the effect of extreme exposure or non-exposure of data on the performance of an application.

6.6 Reuse Analysis Based Approach

In this section we explore the middle ground between the AED and NED schemes. The primary question we are interested in answering is this: "How much performance can we save if we are allowed to expose some portion of the application data during execution?". More specifically, we study a generic scheme which does not employ encryption for writes with reuse distances of δ or less (where δ is a tunable parameter). Our goal here is to reduce the performance overhead by avoiding encrypting data with short reuse distance.

For the purpose of our discussion, a *reuse* is said to occur when a data block in the SPM is written out of the SPM (by a *SPM_wr_back instruction*) and read again into the SPM (through a subsequent *SPM_rd_into* instruction). The key concept

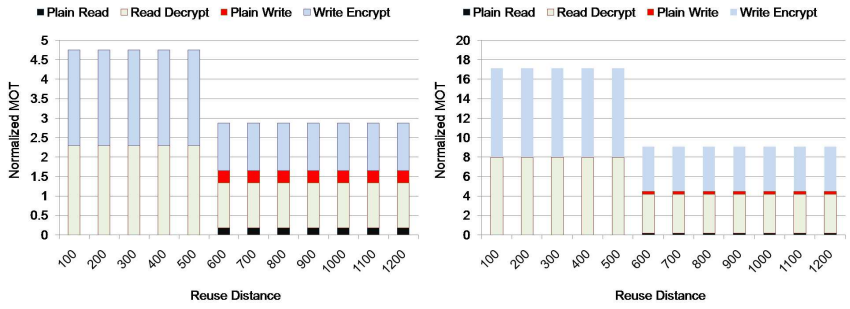
in data reuse analysis is that of *reuse distance* which may be explained as follows. When a block is reused through two memory instructions, the reuse distance is the number of intervening memory instructions between these two memory instructions. Figure 6.9 shows the distribution of successive reuse instructions according to the reuse distances between them. The X-axes represent the number of intervening memory instructions between two memory instructions that reuse a data block. The Y-axes, on the other hand, represent the percentage of instructions amongst all memory instructions that exhibit reuse. Therefore, a bar in the graph at X' in the X-axis of magnitude Y' in the Y-axis indicates that Y' % of memory instructions that exhibit reuse have X' intervening memory instructions between them. It can be seen that the benchmarks *adi* and *tsf* show reuse at very specific values of reuse distance. In comparison, both *lu* and *swim* exhibit reuse across a wider range. [1]

As a write instruction and a read instruction can at most be separated by the rest of the instructions in the trace, the maximum value of the reuse distance possible in an application for two memory instructions is bounded by the total number of dynamic instructions recorded in the memory trace.

As the different applications show differing reuse behavior with respect to the distance, we implemented an *oracle* mechanism, that determines whether the data written by a write instruction in the trace is reused within a certain reuse distance. The trace is formed based on the SPM reuse policy explained in Section 6.3. It allows us to "see" future reuse, because, for any write instruction, the future instructions are those instructions that follow it in the trace.

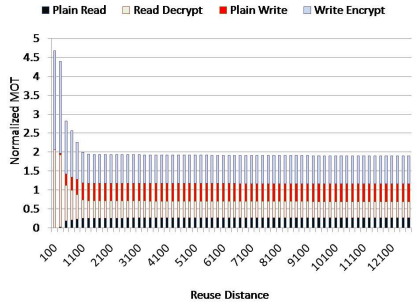
For every write instruction in the trace, the oracle decides whether to encrypt the data while writing it back by examining future read and write instructions upto a specified reuse distance called δ in the trace. This decision is made by the following logic. If the current *SPM_write* instruction is to write a block, and if the same block is read into the SPM within the next δ instructions in the trace, the current write instruction is changed to a *plain-write* instruction. This causes the

[1] Reuse in this scenario means that the data blocks that are written out of the SPM are read back again into the SPM. More conventionally, data reuse is used to refer to the reuse of data within memory, cache or SPM [40]. As the SPM management policy in this work is tuned towards improving performance, it tries to reduce the number of transfers between the off-chip memory and the SPM, by performing all the computations that are needed on a data block (to the extent allowed by data dependencies) when that data block is read into the SPM.

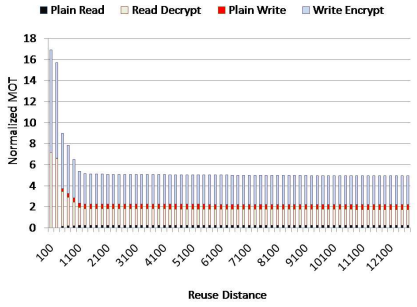


(a) MOT for adi (AES)

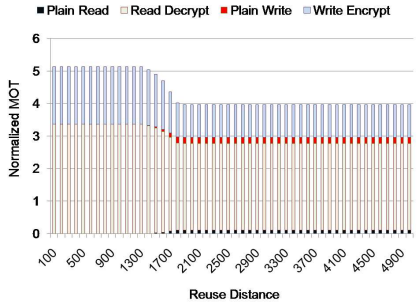
(b) MOT for adi (DES)



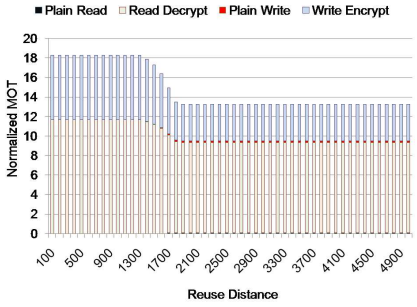
(c) MOT for lu (AES)



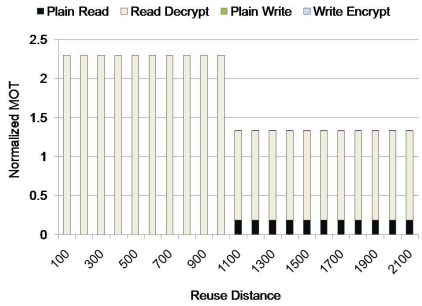
(d) MOT for lu (DES)



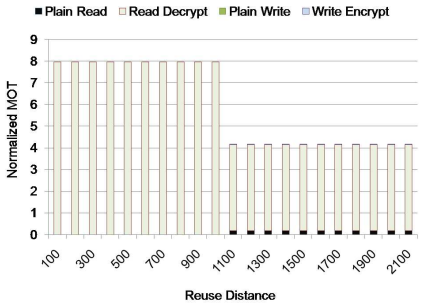
(e) MOT for swim (AES)



(f) MOT for swim (DES)



(g) MOT for tsf (AES)



(h) MOT for tsf (DES)

Figure 6.10. *MOT* values across various reuse distances for our data reuse (oracle) based approach to confidentiality. For each bar in the *MOT* graph, the portion form top to bottom are Write Encrypt, Plain Write, Read Decrypt and Plain Read.

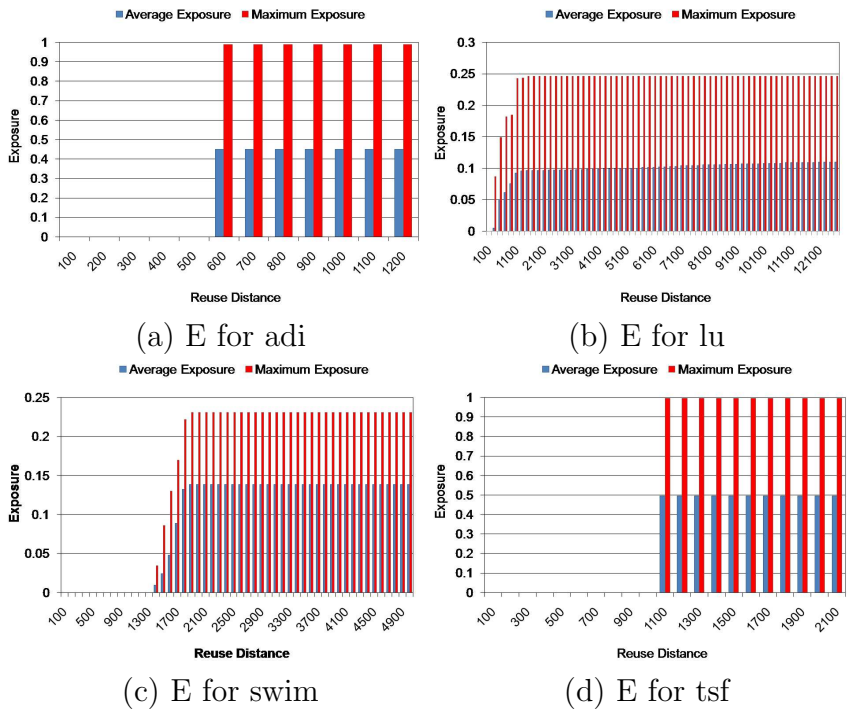


Figure 6.11. E values across various reuse distances for our data reuse (oracle) based approach to confidentiality.

cryptographic engine to set the status of that block as unencrypted and write it to the off-chip memory. In the future, when the *SPM_read* instruction is issued to read that block, the arbiter (see Figure 6.4) sees that it is in fact unencrypted and passes it on directly to the SPM, and simultaneously sends it to the cryptographic engine for encryption. This is called a *plain-read*. All write operations that are not detected to have reuse within a distance of δ are issued as *write-encrypt* instructions and will be treated as *read-decrypts* by the arbiter.

The MOT of this data reuse based scheme is dependent on the total number of *read-decrypts*, *write-encrypts*, *plain-reads* and *plain-write* instructions. The AE and ME values are calculated by measuring the amount of data that is in unencrypted form in the off-chip memory at anytime during the trace (execution). As the amount of data reuse that is observed is dependent on the number of instructions that the oracle can observe, the value of δ is varied from 100 to the number of instructions in the application and the corresponding values of MOT, AE and ME are measured. The value of MOT for all values of δ across the different benchmarks is plotted in Figure 6.10. Similarly, the values of AE and ME for all

values of δ across the different benchmarks are plotted in Figure 6.11.

We see that, for all the benchmarks, there is a variation in the value of MOT, AE and ME. The benchmark *swim* shows a change in the values of MOT, AE and ME over a range of δ. This is because, *swim* exhibits data reuse over a range of reuse distances. On the other hand, *adi* and tsf display data reuse at a very particular value of reuse distance. Hence, the change in values of MOT, AE and ME for *adi* and tsf is abrupt. *lu* shows a drop in MOT and a rise in AE and ME for relatively small values of δ. This is because *lu* exhibits considerable data reuse for small values of δ. In general, it can be observed that there exists a correlation between a drop in the value of MOT and a rise in the values of AE and ME. The extent of this correlation will be examined in the following section.

These results, from the point of view of an application designer, display pertinent information about the effect of the δ parameter on performance and data exposure. However, it may be difficult to gauge from the above graphs which value of δ is most suited for a given application. That is, it may not be very easy for an application designer to tradeoff directly between performance and data vulnerability using the above results. For this reason, a combined metric is proposed in the next section.

6.7 Combined Metric

To arrive at a combined metric we first start by defining two auxiliary metrics to account for both MOT and AE. The first is the *Performance Metric* and the second is called the *Security Metric*. For any specific value of δ, say δ_k, the performance metric is defined as the ratio of the MOT value of the most secure case (value with lowest value of δ) to the MOT with δ_k as the reuse distance. Therefore, the performance metric in effect measures the increase in performance afforded by increasing the reuse distance. Note that, the performance metric is a value that is greater than or equal to 1. The security metric, on the other hand, is defined for a reuse distance, δ_k, as the ratio of the portion secure (unexposed) data when a reuse distance of δ_k is used, to the portion of secure data in the most unsecure case (with the highest value of δ). The security metric measures the benefit to the security of data stored in memory when a lower value of reuse

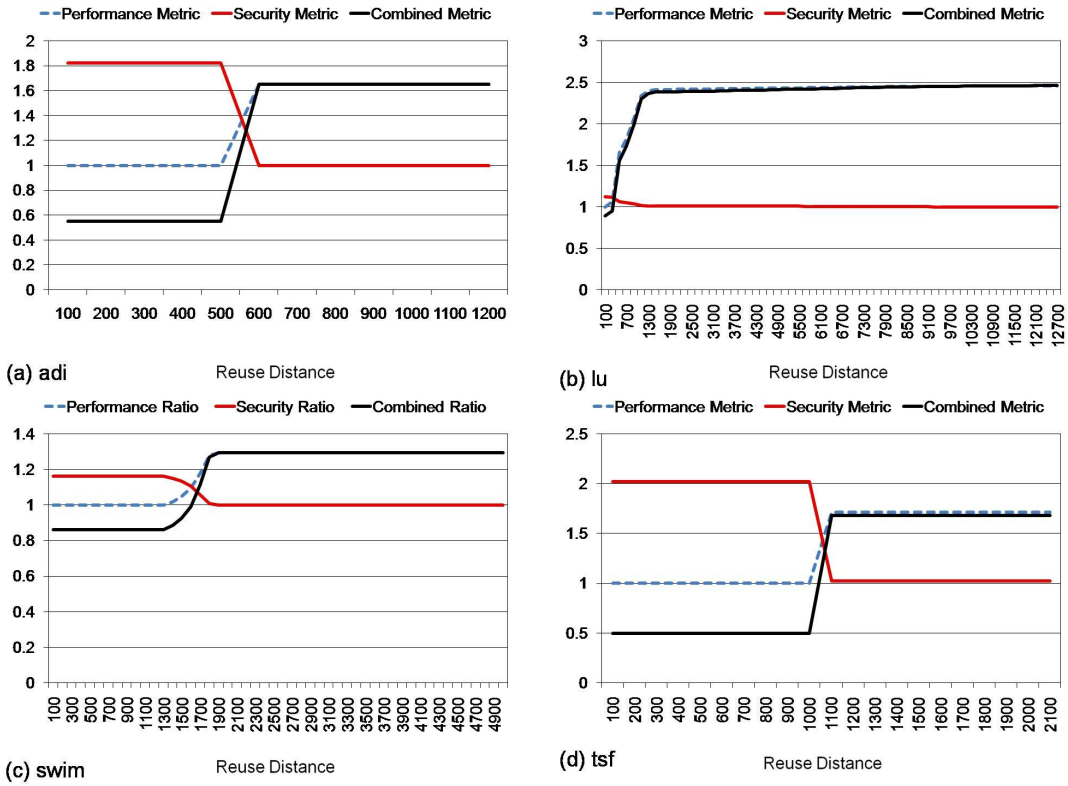


Figure 6.12. The combined metric for different values of reuse distance.

distance is used compared to the most unsecure case. The security metric is also a value that is greater than or equal to 1.

Using the performance metric and the security metric, a *Combined Metric* for a reuse distance δ_k can be defined as its performance metric divided by its security metric. The combined metric therefore represents the *unit gain* in the performance metric for every *unit loss* in the security metric. If the combined metric is less than 1, it means that the performance gained by using δ_k is not proportionate to the increased exposure of data. A combined metric of exactly 1 means that the reuse distance at which this occurs, say δ_k, allows the application to gain in terms of performance what is lost in terms of data confidentiality. Clearly, a preferable case is then to have a combined metric value greater than 1, as this indicates that the δ_k value of reuse distance for which this combined metric is observed allows the application to display performance gain that is disproportionately larger than the loss in confidentiality.

The combined metric is computed for different values of δ for our benchmarks,

and the results are presented in Figure 6.12. It is observed that for *adi*, tsf, and *swim*, the value of the combined metric is below 1 upto a certain reuse distance, beyond which it rises above 1 and, after that it reaches its maximum value. Further increase in the reuse distance does not improve the value of the combined metric. Let us discuss the *swim* results in more detail. Until δ is 1610, the combined metric is below 1. Therefore, an application designer looking to use the data reuse based approach, is likely not to use a value of δ less than 1610. If the value of δ is greater than 1610, the increase in the vulnerability of data is more than offset by the performance gain. Another important point to note is that the value of the combined metric peaks when $\delta = 1700$; beyond this value, there is no rise in the value of the combined metric. Hence, application designers would want to use a value of δ between 1610 and 1700. However, the choice of δ depends on the requirements the application designer has to meet. That is, if the application designer can afford to expose only a small amount data, then she may choose a value of δ for which the combined metric is less than 1. The combined metric for the *lu* benchmark becomes greater than 1 for a small value of reuse distance. Also, it peaks at a much higher value of δ. However, the rate of rise in combined metric with increase in δ is very low and an application designer may not choose to employ high value of δ.

Note that the choice of δ is uniform throughout the code. It is possible to envision that, in a code that comprises of multiple functions that access large amounts of non-intersecting data sets, the value of δ may be varied across these functions. The benchmarks studied in this work do not fall into this category of applications and consequently we do not discuss this possibility further. We observed that it was difficult to separate any benchmark trace into regions based on reuse distances. This is because, in the benchmarks we studied, the read-write instruction pairs of differing reuse distances, occurred in the traces in complex intermingled patterns.

The value of δ chosen by the application designer limits the number of future dynamic instructions that are examined at any time. However, this information is usable by an application designer only if she can modify the static code itself. That is, it should be possible to generate code that achieves the same MOT and AE values without requiring each dynamic write instruction to be individually

Benchmark	**Combined Metric** $= 1$			
	δ	**TYPE 1**	**TYPE 2**	**TYPE 3**
ADI	525	60.0 %	0.0 %	40.0
SWIM	1610	71.88 %	0.0 %	28.12
MxM	100	100.0 %	0.0 %	0.0
TSF	1100	60.0 %	40.0 %	0.0
LU	500	46.66 %	26.67 %	26.67
Benchmark	**Maximum Combined Metric** (> 1)			
	δ	**TYPE 1**	**TYPE 2**	**TYPE 3**
ADI	600	60.0 %	40.0 %	0.0 %
SWIM	1700	71.88 %	28.12 %	0.0 %
MxM	100	100.0 %	0.0 %	0.0 %
TSF	1100	60.0 %	40.0 %	0.0 %
LU	121000	46.66 %	26.67 %	26.67 %

Figure 6.13. Characterization of static instructions according to three types.

marked as a *plain-write* or a *write-encrypt*. In order to achieve this, it should be possible to reason whether each static write instruction in the code translates to a *plain-write* or a *write-encrypt* instruction dynamically; that is, whether a static write always translates to a *plain-write*, or always translates to *write-encrypt*, or can lead to instances of both during execution.

6.8 Profile Based Approach

The analysis in Section 6.7 allows an application designer to decide the best value of δ for a particular application. This allows the application designer to specify the number of future dynamic memory instructions to examine in the memory trace in order to decide whether an SPM block should be written back to memory in an encrypted form or in plain form. Therefore, she can specify each dynamic write instruction to either be a *write-encrypt* or a *plain-write*. However, an oracle based system cannot be implemented using only the original static application code. In this section, we study how it is possible to modify (restructure) the *write* instructions in the static code from the results previously presented and generate a modified application code that mimics the behavior of the reuse based oracle approach. The first step in our profile based approach is to identify the mapping between static instructions in the code and the dynamic instructions in

the memory trace. To do so, each *static-write* instruction is tagged with an id. We then categorized each individual write instruction on the basis of whether it translates into *write-encrypt* or a *plain-write* dynamic instruction. Figure 6.13 presents the results of our profiling. We classified each *static-write* instruction depending on whether it always appears as a *plain-write* (TYPE 1), always as a *write-encrypt* (TYPE 2), or whether it appears sometimes as a *plain-write* and sometimes as a *write-encrypt* instruction (TYPE 3). We present results for two cases. First, when the combined metric is the highest and, second, when it is 1.

When the value of the combined metric is maximum, all the *static-write* instructions in the *adi*, *lu*, *swim*, and *tsf* benchmarks can be classified as either *plain-writes* or *write-encrypts*. This means that, in these applications, it is possible to mark all the write instructions in the code as either *plain-writes* or *write-encrypts* and achieve the same behavior as that achieved by the oracle method with the ideal value of δ. The write instructions in *lu* however have static write instructions that are sometimes issued as *plain-reads* and sometimes and *write-encrypts*. When the value of the *combined metric* is 1, we see that for the benchmarks *adi*, *swim*, *lu*, there exist *static-write* instructions that are classified as TYPE 3.

There are two reasons that contribute to TYPE 3 write instructions: (a) Some of the dynamic instructions that are created by particular static instruction during execution are reused by read instructions while others are not reused, and (b) Part of the reuse is not captured for a particular value of δ. The rest of this section explores how to duplicate the select *static-write* instructions in the code such that one copy produces *plain-writes* and another produces *write-encrypts* through a known code restructuring technique called loop splitting [40].

To use loop splitting for our purposes, we identify all the *static-write* instructions that reuse the data written by a particular TYPE 3 instruction. For each such instruction pair, as we are dealing with array based benchmarks, we identify the loop iteration groups that lead to reuse and those that do not. Then, we isolate (observing data dependences) the two groups as two different loops nests and included the appropriate static instructions *static-plain-write* or *static-write-encrypt* in them. An example of loop splitting is given in Figure 6.14. In Figure 6.14(a) we see that the second loop nest reuses, only part of the data written into array A. Specifically from index 0 to index $(N/2 - 1)$. Therefore, the first loop is split

into two. The loop iterations that exhibits reuse are clubbed together, and those that do not are clubbed together in another loop as shown in Figure 6.14(b). Note that, this loop splitting allows the *static-write* statement in the first loop to be classified as a *static-plain-write* and the second *static-write* instruction as a *static-write-encrypt*. By adding the classification of whether a *static-write* instruction is a *static-plain-write* or *static-write-encrypt* to the *SPM_wr_back* instructions (see Figure 6.3), we achieve the same MOT, ME and AE values as the oracle based method. We use the Omega Library [44] to perform such analysis and generate the appropriate code. The Omega Library is a tool to manipulate integer tuple relations and sets. An integer k-tuple is a point in the space $\mathbb{Z}^k$ and an integer tuple relation is a mapping from tuples to tuples. These relations can be expressed by Presburger formulas [45] which in turn can be constructed by combining affine constraints on integer variables with the logical operations $\neg$, $\vee$, and $\wedge$, and the quantifiers $\forall$ and $\exists$. We express the reuse of data blocks using integer relations (which are in turn expressed by Presburger formulas) between the iterations of the loop nests and the data blocks. The Omega Library is then used to solve these relations to identify the iterations that exhibit reuse and those that do not.

6.9 Sensitivity Analysis

The definition of the security metric is inherently dependent on the metric used to capture the loss of data confidentiality. So far, we have used average and maximum exposure as the metric of data confidentiality. However, an application designer may be interested in using other metrics to capture the confidentiality of her data. To test the sensitivity of the proposed approach to the metric used, we defined two new confidentiality metrics as follows :

• Peak Exposure Duration - This metric defines confidentiality in terms of the maximum time any application datum remains in unencrypted form in memory. A higher value indicates less confidentiality. Figure 6.15(a) shows the Peak Exposure Duration when the NED policy is used.

• Portion of Data Exposed - This metric defines confidentiality in terms of the amount of unique data that is left unencrypted even once. A higher value indicates lesser confidentiality. This metric is different from the Average and Maximum

```
void sum_rows(int A[N], int B[N][N]){
  int C[N/2];
  for(i=0; i < N; i++){
    for(j=0; j < N; j++){
      A[i] += B[i][j];
    }
  }
  for(i=0; i < N/2; i++){
    C[i] = A[i];
  }
}
```

(a)

```
void sum_rows(int A[N], int B[N][N]){
  int C[N/2];
  for(i=0; i < N/2; i++){
    for(j=0; j < N; j++){
      A[i] += B[i][j];
    }
  }
  for(i=N/2; i < N; i++){
    for(j=0; j < N; j++){
      A[i] += B[i][j];
    }
  }
  for(i=0; i < N/2; i++){
    C[i] = A[i];
  }
}
```

(b)

Figure 6.14. (a) Original code in which half of array A is reused. (b) Code after loop splitting in which the iterations that write into the reused portion of array A are separated.

Exposure metrics because, they do not consider the uniqueness of data that is exposed. Figure 6.15(b) shows the Portion of Data Exposed when the NED policy is used.

The security metrics and combined metrics corresponding to the new confidentiality metrics were calculated. Figures 6.15(c) and 6.15(d) show the values of the security and combined metric values for Peak Exposure Duration and Portion

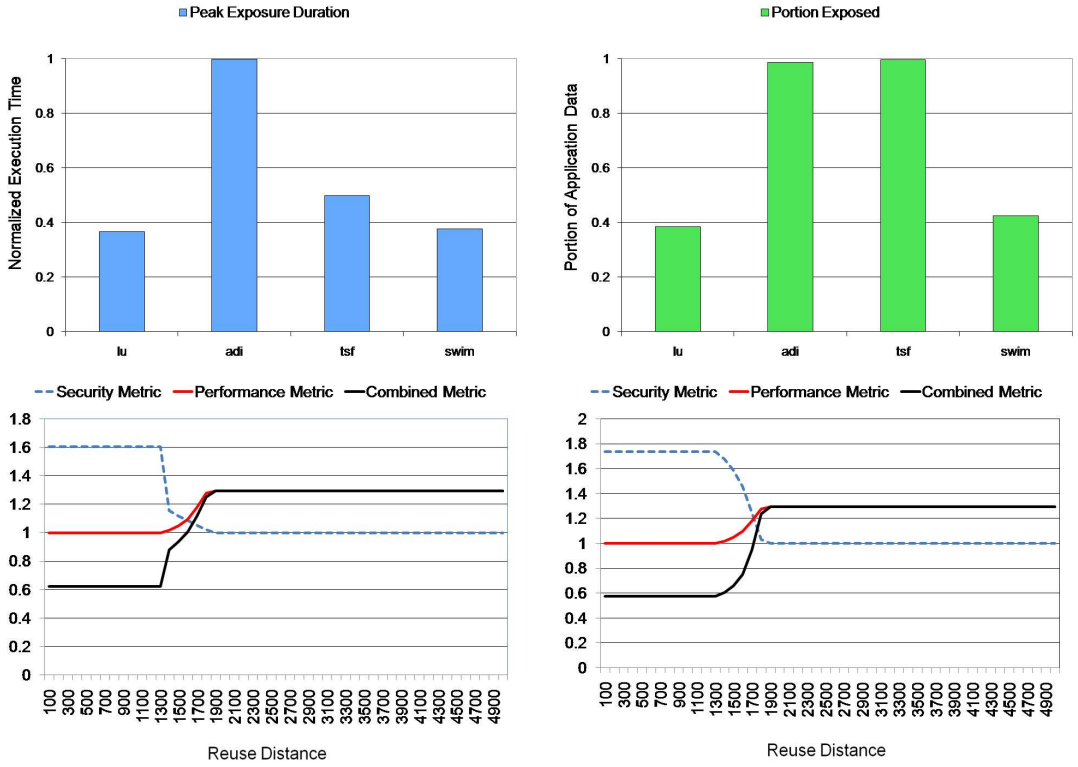


Figure 6.15. Sensitivity results using two alternate confidentiality metrics.

of Data Exposed respectively for the *swim* benchmarks. It can be observed that the trends across the two graphs are very similar to the trends observed using Maximum Exposure (see Figure 6.12(c)) as the confidentiality metric. Thus, for the new metrics, it is possible to calculate the appropriate value of reuse distance and perform profile based analysis to statically generate the required modified application.

In certain scenarios, portions of the data maybe highly sensitive and must not be exposed. The application designer may specify the portions of the data that should never be exposed. Our approach is extendable to this scenario as well.

6.10 Related Work

Due to a lack of space we do not discuss the prior SPM related work in detail [82, 83, 84, 85, 86, 87, 88, 76]. Our approach is different from these efforts in that we focus on the security aspect of an SPM based system. Reducing the

overheads of latency due to security mechanisms employed by computing systems is a problem that has received considerable attention in recent years. One method to improve the latency is to employ encryption algorithms with lesser overhead. However, encryption algorithms are generally used according to the standards that a system is required to support [72]. One Time Pads decouple the cryptographic computations from the corresponding data access [89, 90]. However, they are not suitable due to a high area overhead [91]. Others aim to improve the performance of secure embedded systems using exploring ways to map the code and the data of security software onto the embedded system [92]. [93] builds intrusion resistant software based on secret sharing.

6.11 Conclusion

We studied the effect, on performance and exposure of data, of relaxing the confidentiality requirement that all data in the untrusted portion of a computing system must be in encrypted form. We examined three approaches, namely, always encrypt decrypt, never encrypt decrypt, and an oracle based reuse approach that is ideal but impractical to implement. We then introduced a combined metric to help application designers make an informed decision about the extent of data exposure to incorporate in their design. We finally showed how to use our scheme statically.

Chapter 7

An ILP Based Approach to Perform Selective Encryption of Application Data

7.1 Introduction

This chapter studies the execution of applications with security concerns that need to complete execution within a certain fixed deadline on SPM (scratch-pad memory) based systems. More specifically, the question we are interested in answering is this: "If the application has to finish its execution before a hard deadline which cannot be achieved due to the overhead of encryption and decryption; Is it possible to expose some portion of the application data during execution and meet the deadline"? Clearly, it may be possible to meet the deadline by simply suspending all encryption and decryption (and this is the best possible execution time), but doing so may lead to an unacceptable loss in data confidentiality. Therefore, we need to lose the minimal amount of data confidentiality while meeting the deadline.

We start by using the *data reuse based* approach presented in Section 6.6 and augment it with an Integer Linear Programming (ILP) model that is used to decide the blocks in memory that should be exposed. We study three metrics that an application designer may use to define data confidentiality. Lastly, we also study the dual of the problem presented which is: "Given a minimum level of required

data confidentiality, what is the minimum (or maximum) time that the application can finish in?"

The importance of this work is that it allows an application designer to maximize the confidentiality of an application while operating under a deadline. If the deadline changes, it is very simple to tune the application differently. As mentioned is Chapter 6, the motivation for employing a software/compiler based approach to this problem is that complete reuse information regarding data blocks accessed by an application can only be gathered by looking at the entire window of execution of the application. This is not possible through hardware methods that can only look at a limited window of memory operations. The intuition is that the greater the reuse information that is available, the more accurate the tunability of the application by the application designer.

Section 7.2 presents an ILP approach to minimizing the exposure of data while meeting performance constraints. Section 7.3 presents the results of our proposed approach and Section 7.4 shows how static code is generated to mimic the ILP approach. The chapter is concluded in Section 7.5.

7.2 ILP Model

This section explains on the ILP model used to decide whether a block should be exposed in main memory and when it should be exposed. In our problem formulation, there exist several issued read and write operations that operate on the data blocks in memory. These operations take a certain amount of time to complete and affect the total execution time. The status of the blocks in memory (encrypted/unencrypted) affects the confidentiality of the data in memory. The aim of the ILP model is to keep the total execution time below a specified hard limit and to maximize the confidentiality of the data under this constraint. The variables used in the ILP model are listed in Table 7.1. The inputs are used to specify, when reads and writes take place, the blocks that on which operations take place as well as reuse information. The inputs are obtained form the memory trace explained in Section 6.3 The objective variables are used to calculate the value of the the security metric that the user is interested in. The decision variables specify the status of the blocks as well as whether a particular memory operation causes

	Notation	Explanation
Input	Issued_Read$_i$	ith operation is a read
	Issued_Write$_i$	ith operation is a write
	Block$_{ib}$	Block b is accessed by operation i
	Future_Read$_i$	Write operation in step i on a block has a corresponding read operation in step $j > i$ on the same block
Decision	Encrypted$_{ib}$	Encrypted/Unencrypted state of block b the start of operation i
	rd$_i$	ith operation is a plain read
	derd$_i$	ith operation is a read decrypt
	wr$_i$	ith operation is a plain write
	enwr$_i$	ith operation is a write encrypt
	Total	The total time taken for execution
Objective	Snapshot	Maximum number of blocks ever exposed at any step
	exp_blocks$_b$	Indicates whether a block is exposed at any step
	Tot_exp_blocks	Total number of blocks that are exposed
	exp_time$_b$	The total time for which each block is exposed
	Tot_exp_time	Total exposure time over all blocks

Table 7.1. Notation used in our ILP model.

data in the memory to change its state from encrypted to decrypted or vice-versa. The variable wr_i is used to indicate that a data is being written to the memory in unencrypted form (*plain-write*). The variable rd_i is used to indicate that read operation takes place on unencrypted data (*plain-read*). Similarly, the variable $enwr_i$ is used to indicate that a data is being written to the memory in encrypted form (*write-encrypt*) and $derd_i$ is used to indicate that read operation takes place on encrypted data (*read-decrypt*). In a sense, the ILP model tries to obtain the values of the decision variables for each operation. The following constraints explain the ILP model in detail.

Initial and Final Block State Constraints. Constraints 7.1 and 7.2 specify that all the blocks are initially in an encrypted state and that after the last issued memory operation all the blocks must be encrypted in memory as well.

$$\forall_{b \in Block} \quad Encrypted(1, b) = 1 \tag{7.1}$$

$$\forall_{b \in Blocks} \quad Encrypted(N, b) = 1 \tag{7.2}$$

Read Constraints. Constraints 7.3 and 7.4 specify whether a particular read operation is to be issued as a *plain-read* or a *read-decrypt* operation. Constraint 7.3 states that an Issued_Read should be issued as a *plain-read* if the block being read is in an unencrypted state. Similarly, Constraint 7.4 states that an Issued_Read should be issued as a *read-decrypt* if the block being read is in an encrypted state. Constraint 7.5 ensures that an Issued_Read operation is either a *plain-read* or a *read-decrypt*.

$$\forall_{i \in Issued\_Ops} \quad rd(i) \geq Issued\_Read(i) * \sum_{b \in Blocks} (1 - encrypted(i, b)) * Block(i, b) \tag{7.3}$$

$$\forall_{i \in Issued\_Ops} \quad derd(i) \geq Issued\_Read(i) * \sum_{b \in Blocks} encrypted(i, b) * Block(i, b) \tag{7.4}$$

$$\forall_{i \in Issued\_Ops} Issued\_Read(i) = rd(i) + derd(i) \tag{7.5}$$

Write Constraints. Constraint 7.6 states that a write operation *may* be issued as a *plain-write* operation, when there exists a future read (reuse) for that block. That is, the block is reused. This constraint in effect is used to store the block in a decrypted state in the memory. The constraint also implies although a write operation may be a *plain-write* if a future read exists for the block; but the existence of a future read need not result in a *plain-write*. The choice is made by the solver depending on the security metrics. Constraint 7.7 ensures that an Issued_Write operation is either a *plain-write* or a *write-encrypt*. Lastly, Constraint 7.8 ensures that the number of *plain-reads* is the same as the number of *plain-writes*.

$$\forall_{i \in Issued\_Ops} \quad wr(i) \leq Issued\_Write(i) * \sum_{b \in Blocks} (Future\_Read(i) * Block(i, b)) \tag{7.6}$$

$$\forall_{i \in Issued\_Ops} Issued\_Write(i) = wr(i) + enwr(i) \tag{7.7}$$

$$\sum_{i \in Issued\_Ops} wr(i) = \sum_{i \in Issued\_Ops} rd(i) \tag{7.8}$$

Block State Propagation Constraints. The state of a block in memory before the current operation depends on the operation performed on that block in the previous step and the status of the block at the start of the previous step. If no operation is performed on a block, then the block should retain the same status. If an operation is performed on a block, it may change its status.

Constraint 7.9 states that a block must be encrypted if it is encrypted in the previous step and there has been no *plain-write* operation on that block in the previous step. Constraint 7.10 indicates that a block must be encrypted if it is encrypted if there was a *plain-read* operation on that block in the previous step. Finally, Constraint 7.11 states that a block must be encrypted if there is an *write-encrypt* operation on that block in the previous step.

$$\forall_{\substack{i \in Issued\_Ops \\ b \in Blocks}} \quad encrypted(i,b) \geq encrypted(i-1,b) - wr(i-1) * Block(i-1,b) \quad (7.9)$$

$$\forall_{i \in Issued\_Ops b \in Blocks} \quad encrypted(i,b) \geq rd(i-1) * Block(i-1,b) \quad (7.10)$$

$$\forall_{i \in Issued\_Ops b \in Blocks} \quad encrypted(i,b) \geq enwr(i-1) * Block(i-1,b) \quad (7.11)$$

Total Execution Time Constraints. Constraint 7.12 calculates the total execution time as a function of the number of *plain-read*, *plain-write*, *read-decrypt* and *write-encrypt* operations. Constraint 7.13 limits the value of the total execution latency to a user specified limit. These two constraints together force the solver to choose plain and encrypt/decrypt operations such that the total execution time limit is met. The constants in front of the operations are the actual machine cycles required to perform each operation for a data block. In this study, we measured the times taken under the AES [73] scheme. The implementation used for AES is from the OpenSSL cryptography toolkit [81]. The time to decrypt (encrypt) a single block is calculated by performing one million decrypt (encrypt) operations and taking the average time over all the operations.

$$Total = \sum_{i \in Issued\_Ops} (169 * rd(i) + 1035 * derd(i) + 281 * wr(i) + 1107 * enwr(i)) \quad (7.12)$$

$$Total \leq ConstantValue \quad (7.13)$$

Objective Function Constraints. The following constraints form the objective functions of the ILP system. They aim to maximize any one of three security metrics that the user is interested in. Figure 7.1 explains the three confidentiality metrics that a user may be interested in. The objective functions can be modified easily to consider other metrics as well.

Objective Function 1. Constraint 7.14 determines the number of blocks exposed at any stage of execution and 7.15 specifies that this amount should be

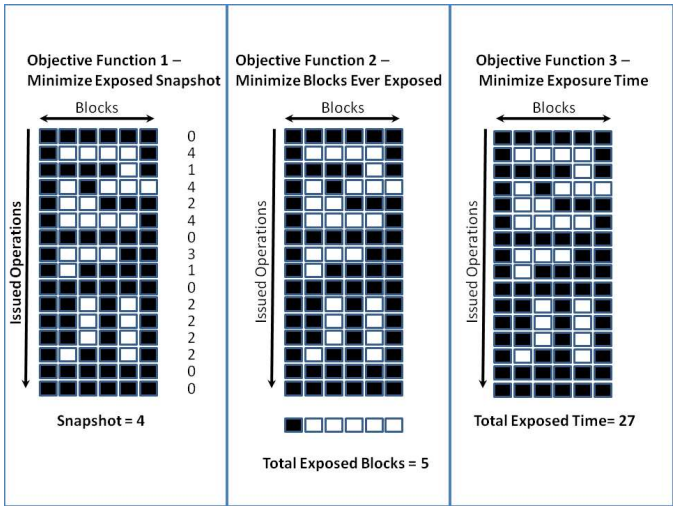


Figure 7.1. Illustration of the three Objective functions studied. Objective 1 minimizes the number of blocks that are exposed at every stage of execution. Objective 2 minimizes the total number of unique blocks that are ever exposed. Objective 3 minimizes the total exposure Time (Blocks * Issued_Ops).

minimized.

$$\forall_{i \in Issued\_Ops} \quad Snapshot >= \sum_{b \in Blocks} (1 - encrypted(i, b)) \tag{7.14}$$

$$Minimize(Snapshot) \tag{7.15}$$

Objective Function 2. Constraint 7.16 determines whether a block is exposed at any stage of execution. Constraint 7.17 calculates the total blocks *ever* exposed across all the steps and 7.18 specifies that this amount should be minimized.

$$\forall_{b \in Blocks i \in Issued\_Ops} exposed\_blocks(b) \geq (1 - encrypted(i, b)) \tag{7.16}$$

$$Total\_exposed\_blocks = \sum_{b \in Blocks} exposed\_blocks(b) \tag{7.17}$$

$$Minimize(Total\_exposed\_blocks) \tag{7.18}$$

Objective Function 3. Constraint 7.19 and calculates the total number of steps for which each block is exposed. Constraint 7.20 calculates the total steps for which all the blocks are exposed and Constraint 7.21 specifies that this amount should be minimized.

$$\forall_{b \in Blocks} \quad exposed\_time(b) = \sum_{i \in Issued\_Ops} (1 - encrypted(i, b)) \tag{7.19}$$

Benchmark	Static Instructions		Dynamic Instructions		Data Transfer
	Reads	Writes	Reads	Writes	Size (MB)
ADI [77]	15	15	576	576	1.68
SWIM [78]	61	19	3319	1642	4.84
TSF [79]	5	5	1025	1020	15.97
LU [80]	13	13	33952	39456	4.48

Figure 7.2. Details of the benchmarks studied.

$$Total\_exposed\_time = \sum_{b \in Blocks} exposed\_time(b) \quad (7.20)$$

$$Minimize(Total\_exposed\_time) \quad (7.21)$$

7.2.1 Dual Model

The dual to the objective functions presented above is to minimize the total execution time, while keeping the confidentiality above a minimal level.

Confidentiality Metric. Constraints 7.22 and 7.23 calculate and limit the number of blocks exposed at any stage of execution (see objective function 1).

$$\forall_{i \in Issued\_Ops} \quad Snapshot >= \sum_{b \in Blocks} (1 - encrypted(i, b)) \quad (7.22)$$

$$Snapshot \leq ConstantValue \quad (7.23)$$

Dual Objective Function. Constraint 7.24 calculates the total execution time as before and constraints 7.25 states that this value should be minimized. It is also possible to study the longest possible execution time by stating in Constraint 7.25 that $Total$ should be maximized. This gives us the range of possible execution times for each application.

$$Total = \sum_{i \in Issued\_Ops} (169 * rd(i) + 1035 * derd(i) + 281 * wr(i) + 1107 * enwr(i)) \quad (7.24)$$

$$Minimize(Total) \quad (7.25)$$

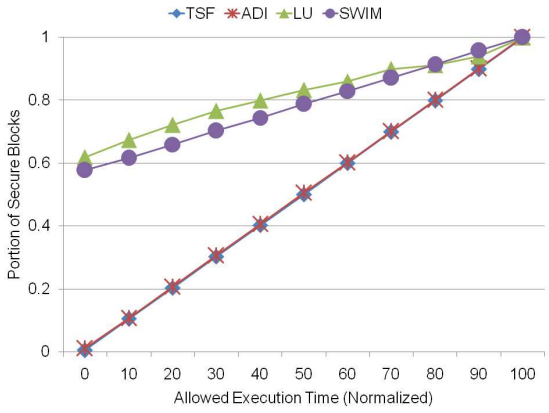


Figure 7.3. The portion of blocks that are secure during execution (Objective 1) across the range of Allowed Execution Times (normalized).

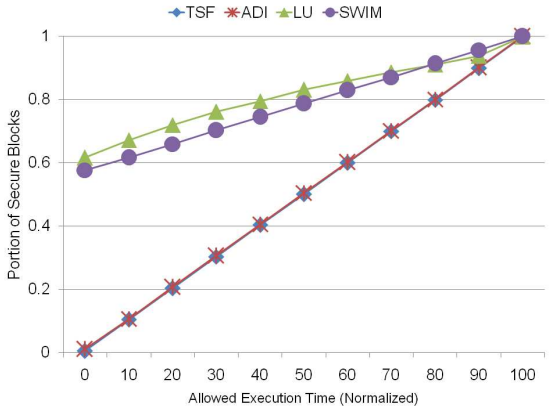


Figure 7.4. The portion of blocks that are never exposed during execution (Objective 2) across the range of Allowed Execution Times (normalized).

7.3 Experimental Evaluation

The results presented here explore how the value of the different security metrics vary for the applications as the amount of execution time allowed is increased. Figure 7.2 presents the details of the applications that are studied. The range of the excess execution time that each application is allowed varies from a minimum, when all possible blocks are kept unencrypted, to a maximum when all the blocks are always encrypted. The duals of the objective functions are used to determine this range for each benchmark and objective function.

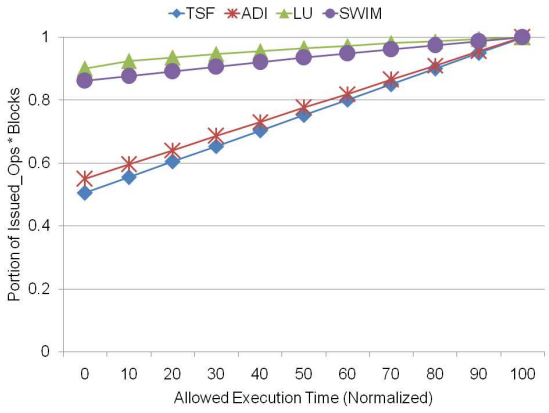


Figure 7.5. The portion of blocks that are never exposed (Objective 3) during execution across the range of Allowed Execution Times (normalized).

The ILP model was coded using XPressMP, a commercial ILP solver [34]. A practical implementation issue was that the memory requirements for large inputs was more that what was available in the machines used to run the solver. This occurred because the number of issued operations for benchmark *lu* and *swim* were large. Therefore, for benchmarks with large number of dynamic instructions, the input was separated at the granularity of groups of blocks. The solver was used to solve the sub problems and the results generated were aggregated. The overhead of the solver was not significant for the benchmarks *adi* and *tsf* where the solver took less than a minute to arrive at a solution. However, for the benchmarks *swim* and *lu* in which the input was divided into sub-parts, the solution took was obtained in a manner of hours when the solver is run sequentially on each sub-part. However, an easy optimization is to simply use multiple processors and take advantage of the coarse grain parallelism that is possible.

Figure 7.3 shows the percentage of blocks that are secure at any point during the execution of the application when the execution time limit is increased through the calculated range. It is clear that, across all the benchmarks, an increase in the stipulated finish time results in an increased portion of the blocks remaining secure per execution step (Objective 1). However, for the different benchmarks, the rate of increase in secure portion is different. Figure 7.4 shows the portion of blocks that are always secure during the execution of the application when the execution time allowed is increased (Objective 2). Finally, Figure 7.5 shows the portion of the

Benchmark	TYPE 1	TYPE 2	TYPE 3
ADI	16.67 %	50.0 %	33.33
SWIM	12.00 %	52.0 %	36.0
TSF	20.0 %	60.0 %	20.0
LU	0.0 %	33.33 %	66.67

Figure 7.6. Characterization of static instructions according to three types for Objective Function 1 (Minimized Snapshot) when the time limit specified is halfway in the possible range (determined by the dual of Objective Function 1).

$Issued\_Ops*Blocks$ product that is secure during the execution of the applications when the total execution time allowed is increased through the range (Objective 3).

Figures 7.3, 7.4 and 7.5 indicate that different objective functions used to express different security concerns lead to different rates of increase in the security metrics of the different applications. This is attributed to the intuitive observation that when the execution time allowed is the highest in the range, all the data is kept encrypted in memory. However, when the the excess execution time allowed is low, the amount of data that can be exposed is bounded by the amount of data reuse in the benchmark. Thus, different benchmarks show different values for the objective functions for lower execution time limits.

An application designer can use the presented results in two ways. Firstly, she can study the rate of increase of the confidentiality of the application through the range of allowed finish time. In particular, changes to the application reuse characteristics through optimizing compilation may lead to different solution spaces (see Figure 6.1). Second, the designer can specify a hard deadline or a hard confidentiality metric and generate code that performs the appropriate data transfer operations.

7.4 Code Generation

The analysis in Section 7.2 determines each dynamic write instruction in the memory trace to either be a *write-encrypt* or a *plain-write* which in turn determines the status of the blocks in memory. We use the profile based approach explained in Section 6.8 to identify the mapping between static instructions in the code and the

dynamic instructions in the memory trace. We classified each *static-write* instruction depending on whether it always appears as a *plain-write* (TYPE 1), always as a *write-encrypt* (TYPE 2), or whether it appears sometimes as a *plain-write* and sometimes as a *write-encrypt* instruction (TYPE 3). Figure 7.6 presents the results of our profiling for Objective Function 1, when the time limit specified is halfway in the possible range as determined by the dual of Objective 1 (see Section 7.2.1). Using the loop-splitting analysis presented in Section 6.8 we were able to create the static code and generate a modified application code that mimics the behavior of the ILP model.

7.5 Conclusion

We have used an ILP based mechanism to maximize the confidentiality of an application running on a scratch pad memory based system, while keeping its execution time below a specified limit. Our approach consisted of three steps : Reuse based analysis, ILP Modeling, and Code Generation. Experiments with four benchmarks have shown that the proposed approach works in practice.

Chapter 8

Future Work

There remain several interesting issues to be tackled in the domain of CPMs and parallel architectures in general. Using the source analysis performed, optimizations must be realized for the execution of multiple threads of an application executing in parallel. This thesis has considered metrics of optimization such as temperature reduction and data confidentiality on an individual basis and ignores the presence of other robustness concerns. This may not be right approach when multiple robustness concerns exist.

In the uni-processor domain, it is well known that as more compiler optimizations are utilized, they may possibly degrade performance than rather than improve it. The best combination of optimizations is chosen through methods such as iterative exploration, combined elimination, genetic programming and the use of performance counters. These methods address the problem solely from from the perspective of performance rather than robustness. Certain systems may require other optimization metrics such as memory usage to be considered as well. A key issue is to understand the effect of the interaction of optimizations targeted at different goals in a multi-core domain. We propose to study approaches to explore the optimization space when multiple roubstness goals exist along with performance concerns.

A short term goal is to extend work in Chapter 6 to a Chip Multiprocessor (CMP). Briefly, Chapter 6 examined the trade-offs between the security offered by maintaining data in memory in encrypted form and the performance gained by not encrypting it. This work was targeted at a uniprocessor system that employed a

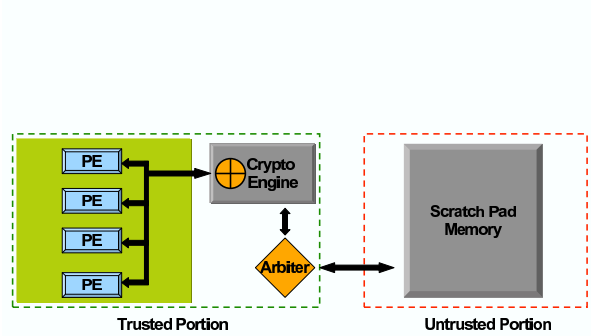


Figure 8.1. The 4 processor CMP used in the proposed work.

Scratch Pad Memory (SPM) that was part of the *trust base*. The aim is to study the trade offs between the security afforded by encryption and the performance lost because of it in a CMP that uses an SPM. The specific secure parallel architecture considered is shown in Figure 8.1. This architecture is a four-processor CMP, in which each processor is connected via a cryptographic engine to the Scratch Pad Memory. In contrast the the work presented in Chapter 6, the SPM is not part of the trust base. That is, the data on the SPM is in encrypted form by default. Data that is brought to the processor is decrypted before it is used by the processor. Similarly, data is encrypted before it is written back to the SPM. As the encryptions and decryptions lie on the critical path of execution, they directly influence the performance of the application. We will investigate the effect of relaxing the requirement that all data outside the trust base should be in encrypted form.

In allowing the data to exist in decrypted form in the SPM, the relative level sensitivity of the data is also an important consideration. This is because the application designer may be willing to expose certain portion of the data but not others as parts of the data may be more important than others. The potential classification scheme of the data is based on the multilevel security model, is shown in Figure 8.2. The application designer should be able to specify the different levels of sensitivity of the data and specify the threshold of sensitivity that she is willing to

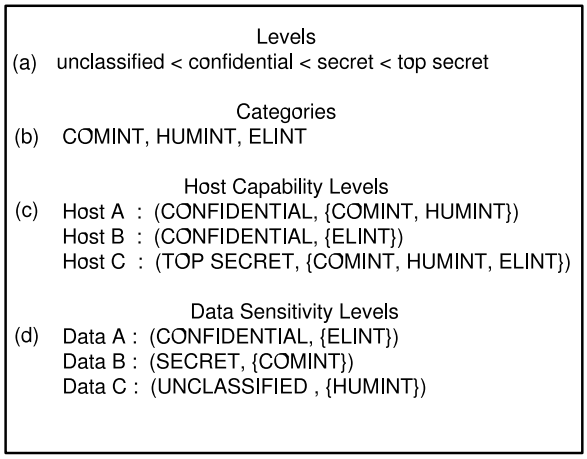


Figure 8.2. Multilevel security. (a) A Lattice of different security levels (b) Categories of data (c) Host capability levels (d) Data sensitivity levels.

expose. For example, the designer may be willing to expose data that is classified as *confidential*, but not *secret* data. A potential pitfall, is that the application designer may not be able to specify the sensitivity of all the data accessed by this application. A possible solution for this problem is a technique proposed in [94]. However, this method will not account for indirect flows of information. A solution for this has to be sought out.

There remain several robustness issues to be tackled. For example, process variation is a phenomenon that could result in identically designed processors operating at different frequencies. Negative bias temperature instability (NBTI) is a phenomenon that could cause the parameters of transistors which results in a reliability issues.

Bibliography

[1] Beygelzimer, A., G. Grinstein, and I. Rish (2004) "Improving Network Robustness," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pp. 322–323.

[2] "http://www.research.ibm.com/autonomic/," .

[3] Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr (2004) "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, **1**(1), pp. 11–33.

[4] Pullum, L. L. (2001) *Software Fault Tolerance Techniques and Implementation*, Artech House.

[5] Tal, M. (2008), "The real cost of application outages," .

[6] Hammond, L., B. A. Nayfeh, and K. Olukotun (1997) "A Single-Chip Multiprocessor," *Computer*, **30**(9), pp. 79–85.

[7] Barroso, L. A., K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese (2000) "Piranha: a scalable architecture based on single-chip multiprocessing," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pp. 282–293.

[8] Burger, D., S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team (2004) "Scaling to the End of Silicon with EDGE Architectures," *Computer*, **37**(7), pp. 44–55.

[9] Nagarajan, R., K. Sankaralingam, D. Burger, and S. Keckler (2001) "A design space evaluation of grid processor architectures," *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 40–51.

[10] Sankaralingam, K., R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore (2003) "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," *SIGARCH Comput. Archit. News*, **31**(2), pp. 422–433.

[11] Taylor, M. B., J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal (2002) "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, **22**(2), pp. 25–35.

[12] Rohou, E. and M. D. Smith (1999) "Dynamically managing processor temperature and power," in *In 2nd Workshop on Feedback-Directed Optimization*.

[13] Heo, S., K. Barr, and K. Asanović (2003) "Reducing power density through activity migration," in *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pp. 217–222.

[14] Hung, W.-L., Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin (2005) "Thermal-Aware Task Allocation and Scheduling for Embedded Systems," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 898–899.

[15] Shang, L., L.-S. Peh, A. Kumar, and N. K. Jha (2004) "Thermal Modeling, Characterization and Management of On-Chip Networks," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 67–78.

[16] Reinhardt, S. and S. Mukherjee (2000) "Transient fault detection via simultaneous multithreading," *SIGARCH Comput. Archit. News*, **28**(2), pp. 25–36.

[17] Michalak, S., K. Harris, N. Hengartner, B. Takala, and S. Wender (Sept. 2005) "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *Device and Materials Reliability, IEEE Transactions on*, **5**(3), pp. 329–335.

[18] Wang, N., J. Quek, T. Rafacz, and S. patel (2004) "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, p. 61.

[19] Patel, J. (2004) "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Trans. Dependable Secur. Comput.*, **1**(2), pp. 128–143.

[20] Degalahal, V., R. Ramanarayanan, N. Vijaykrishnan, Y. Xie, and M. J. Irwin (2004) "The Effect of Threshold Voltages on the Soft Error Rate," in *International Symposium on Quality Electronic Design*, pp. 503–508.

[21] Brooks, D. and M. Martonosi (2001) "Dynamic Thermal Management for High-Performance Microprocessors," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, p. 171.

[22] Donald, J. and M. Martonosi (2004) "Temperature-Aware Design Issues for SMT and CMP Architectures," in *Workshop on Complexity-Effective Design*.

[23] "http://www.hpl.hp.com/research/dca/smart_cooling/," .

[24] Kadayif, I., M. Kandemir, and M. Karakoy (2002) "An energy saving strategy based on adaptive loop parallelization," in *DAC '02: Proceedings of the 39th conference on Design automation*, pp. 195–200.

[25] Skadron, K., T. Abdelzaher, and M. R. Stan (2002) "Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management," in *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, p. 17.

[26] Skadron, K., M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan (2003) "Temperature-Aware Microarchitecture," *International Symposium on Computer Architecture*, **0**, p. 2.

[27] Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Sheipman, and D. Wonnacott (1996), "The Omega calculator and library, version 1.1.0." .

[28] Pugh, W. (1994) "Counting solutions to Presburger formulas: how and why," in *Proceedings of PLDI*, pp. 121–134.

[29] Chaparro, P., G. Magklis, J. Gonzalez, and A. Gonzalez (2005) "Distributing the Frontend for Temperature Reduction," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 61–70.

[30] Narayanan, S. H. K., G. Chen, M. Kandemir, and Y. Xie (2005) "Temperature-Sensitive Loop Parallelization for Chip Multiprocessors," in *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pp. 677–682.

[31] Benini, L. and G. D. Micheli (2002) "Networks on chips: a new SoC paradigm," *Computer*, **35**(1), pp. 70–78.

[32] Dally, W. J. and B. Towles (2001) "Route packets, not wires: on-chip inteconnection networks," in *DAC '01: Proceedings of the 38th conference on Design automation*, pp. 684–689.

[33] Hu, J. and R. Marculescu (2003) "Energy-aware mapping for tile-based NoC architectures under performance constraints," in *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, pp. 233–239.

[34] Guret, C. et al. (2000) *Applications of Optimization with XpressMP*, Dash optimization Ltd.

[35] Avizienis, A. (1978) "On the implementation of nversion programming for software fault tolerance during execution," *Proceedings of the IEEE*, **66**(10), pp. 1109–1125.

[36] Elmendorf, W. (1972) "Fault-tolerant programming," in *FTCS-2*, pp. 79–83.

[37] Randell, B. (1975) "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering*, **SE-1**(2), pp. 220–232.

[38] Horning, J. J. and al. (1974) "A program structure for error detection and recovery," in *Operating Systems, Proceedings of an Int. Symposium*, Springer-Verlag, pp. 171–187.

[39] Pullum, L. (1993) "A new adjudicator for fault tolerant software applicationscorrectly resulting in multiple solutions," in *Digital Avionics Systems Conference*, pp. 147–152.

[40] Wolfe, M. (1996) *High Performance Compilers for Parallel Computing*, Addison-Wesley.

[41] Wolfe, M. J. (1990) *Optimizing Supercompilers for Supercomputers*, MIT Press.

[42] Kodukula, I. and al. (1997) "Data-centric multi-level blocking," in *PLDI*, pp. 346–357.

[43] Kadayif, I. and M. Kandemir (2005) "Data space-oriented tiling for enhancing locality," *Trans. on Embedded Computing Sys.*, **4**(2), pp. 388–414.

[44] Kelly, W. and al. (1996) *The Omega Calculator and Library v1.1.0*, *Tech. rep.*, Dept. of CS, Univ. of Maryland.

[45] Kreisel, G. and J. L. Krivine (1967) *Elements of mathematical logic*, North-Holland Pub. Co.

[46] Chen, C. and M. Hsiao (1992) "Error-correcting codes for semiconductor memory applications: a state of the art review." *Reliable Computer Systems - Design and Evaluation*, pp. 771–786.

[47] Pradhan, D. K. (ed.) (1996) *Fault-tolerant computer system design*.

[48] Kelly, W. and al. (1994) *Code generation for multiple mappings*, *Tech. rep.*, Dept. of CS, Univ. of Maryland.

[49] Gurumurthi, S., A. Parashar, and A. Sivasubramaniam (2005) "SOS: Using Speculation for Memory Error Detection," in *Workshop on High Performance Computing Reliability Issues*.

[50] Foster, I., N. Karonis, C. Kesselman, and S. Tuecke (1998) "Managing security in high-performance distributed computing," *Cluster Computing*, **1**, pp. 95–107.

[51] Kocher, P., R. Lee, G. McGraw, and A. Raghunathan (2004) "Security as a new dimension in embedded system design," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, pp. 753–760.

[52] Zdancewic, S., L. Zheng, N. Nystrom, and A. C. Myers (2001) "Untrusted hosts and confidentiality: secure program partitioning," *SIGOPS Oper. Syst. Rev.*, **35**(5), pp. 1–14.

[53] Bell, D. E. and L. J. LaPadula (1973) *Secure Computer Systems: Mathematical Foundations*, *Tech. Rep. 2457*.

[54] Zheng, L., S. Chong, A. C. Myers, and S. Zdancewic (2003) "Using Replication and Partitioning to Build Secure Distributed Systems," in *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, p. 236.

[55] Neumann, P. G., R. J. Feiertag, K. N. Levitt, and L. Robinson (1976) "Software development and proofs of multi-level security," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pp. 421–428.

[56] Myers, A. C. and B. Liskov (2000) "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, **9**(4), pp. 410–442.

[57] Karger, P. A. (2005) "Multi-Level Security Requirements for Hypervisors," in *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pp. 267–275.

[58] Maret, S. (2005), "On Their Own Terms: A Lexicon with an Emphasis on Information-Related Terms Produced by the U.S. Federal Government," .

[59] Thulasiraman, K. and M. Swamy (1992) *Graphs : theory and algorithms*.

[60] Kreisel, G. and J. L. Krivine (1967) *Elements of mathematical logic*, North-Holland Publishing Company.

[61] Culler, D. E., J. P. Singh, and A. Gupta (1999) *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers.

[62] Denning, D. E. and P. J. Denning (1977) "Certification of programs for secure information flow," **20**(7), pp. 504–513.

[63] "http://www.cs.cornell.edu/jif/," .

[64] "http://www.cs.umd.edu/ jfoster/cqual/," .

[65] Loscocco, P. and S. Smalley (2001) "Integrating Flexible Support for Security Policies into the Linux Operating System," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–42.

[66] Douglis, F., M. F. Kaashoek, J. K. Ousterhout, and A. S. Tanenbaum (1991) "A comparison of two distributed systems: Amoeba and sprite," *ACM Transactions on Computer Systems*, **4**(4).

[67] Zhang, X. and R. Gupta (2003) "Hiding program slices for software security," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pp. 325–336.

[68] Livshits, V. B. and M. S. Lam (2003) "Tracking pointers with path and context sensitivity for bug detection in C programs," *SIGSOFT Softw. Eng. Notes*, **28**(5), pp. 317–326.

[69] Fields, G. (2006) "Stolen Disk Leads to Fears Of VA Privacy Breach," *The Wall Street Journal*.

[70] ABC News (2007), "Hackers Steal 1.6 Million Files From Monster.com," .

[71] cbsnews.com (2005), "Hackers Hit Lexis Nexis Database," .

[72] Ravi, S., A. Raghunathan, P. Kocher, and S. Hattangady (2004) "Security in embedded systems: Design challenges," *Trans. on Embedded Computing Sys.*, **3**(3), pp. 461–491.

[73] Nechvatal, J. et al (2001) "Report on the Development of the Advanced Encryption Standard (AES)," *Journal of Research of the National Institute of Standards and Technology*, **106**, pp. 511–576.

[74] Merkle, R. C. and M. E. Hellman (1981) "On the security of multiple encryption," *Commun. ACM*, **24**(7), pp. 465–467.

[75] Kandemir, M. et al (2001) "Dynamic management of scratch-pad memory space," in *DAC*, pp. 690–695.

[76] Udayakumaran, S. et al (2006) "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Sys.*, **5**(2), pp. 472–511.

[77] McMahon, F. (1988) "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," , pp. 143–186.

[78] KleinOsowski, A., J. Flynn, N. Meares, and D. J. Lilja (2001) "Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research," , pp. 83–100.

[79] Berry, M. and al. (1989) "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int. Journal of Supercomputing Applications*, **3**(3), pp. 5–40.

[80] Womble, D. et al (1994) *LU Factorization and the LINPACK Benchmark on the Intel Paragon*, *Tech. Rep. SAND94-0425*, Sandia National Laboratories.

[81] http://www.openssl.org/, "OpenSSL," .

[82] Francesco, P. et al (2004) "An integrated hardware/software approach for run-time scratchpad management," in *DAC*, pp. 238–243.

[83] Janapsatya, A. et al. (2004) "Hardware/software managed scratchpad memory for embedded system," in *ICCAD*, pp. 370–377.

[84] Absar, M. J. and F. Catthoor (2005) "Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access," in *DATE*, pp. 1162–1167.

[85] Dominguez, A. et al (2007) "Recursive function data allocation to scratch-pad memory," in *CASES*, pp. 65–74.

[86] Baiocchi, J. et al (2007) "Fragment cache management for dynamic binary translators in embedded systems with scratchpad," in *CASES*, pp. 75–84.

[87] Nguyen, N. et al (2005) "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *CASES*, pp. 115–125.

[88] ET AL, M. V. (Oct. 2006) "Cache-Aware Scratchpad-Allocation Algorithms for Energy-Constrained Embedded Systems," *Trans. on Computer-Aided Design of Integrated Circuits and Systems*, **25**(10), pp. 2035–2051.

[89] SHI, W. ET AL (2005) "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," *SIGARCH Comput. Archit. News*, **33**(2), pp. 14–24.

[90] SUH, G. E. ET AL (2003) "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *MICRO*, pp. 339–350.

[91] NAGARAJAN, V., R. GUPTA, and A. KRISHNASWAMY (2007) "Compiler-Assisted Memory Encryption for Embedded Processors," in *HiPEAC*, pp. 7–22.

[92] ARORA, D. ET AL (2006) "Software architecture exploration for high-performance security processing on a multiprocessor mobile SoC," in *DAC*, pp. 496–501.

[93] ZHANG, T. ET AL (2005) "Building Intrusion-Tolerant Secure Software," in *CGO*, pp. 255–266.

[94] LI, F., M. KANDEMIR, R. BROOKS, and G. CHEN (2005) "A Compiler-Based Approach to Data Security," in *International Conference on Compiler Construction*, pp. 188–203.

Vita

Sri Hari Krishna Narayanan

348D IST building
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802 USA

Phone (814) 574-5334
Fax: (814) 865-3176
snarayan@cse.psu.edu
www.cse.psu.edu/~snarayan

RESEARCH INTERESTS

Compiler optimizations for emergent architectures, Robust computing.

EDUCATION

Doctoral Candidate, Dept. of Computer Science and Engineering, 2003-Present
The Pennsylvania State University, University Park, PA, USA.

Bachelor of Engineering, Computer Science and Engineering, 1999 – 2003
Sri Venkateswara College of Engineering, University of Madras, Chennai, India.

EXPERIENCE

Graduate Assistant, Penn State Aug 2003 - Apr 2007, Jan 2008 - Jun 2008
As a research assitant worked in the Microsystems DEsign Laboratory. As teaching ssistant, conducted recitation, held office hours and graded assignments for undergraduate courses of about 120 students.

Teaching Intern, Penn State Sep 2007 - Dec, 2007
Co-taught CSE 331- Computer organization and design with Dr. Mary Jane Irwin. Responsibilities included conducting lectures, designing programming assignments and creating examinations.

Givens Associate, Argonne National Laboratory May, 2007 - August, 2007
Supervised by Dr. Boyana Norris. Coded a software performance estimation tool. Worked on the development of ADIC, an automatic differentiation tool for C programs. In particular, contributed towards extending ADIC to handle C++ input code.

SELECTED PUBLICATIONS

1. Sri Hari Krishna Narayanan, Mahmut Kandemir, Richard Brooks, Performance Aware Secure Code Partitioning, in the *proceedings of DATE 2007.*

2. Sri Hari Krishna Narayanan, Guilin Chen, Mahmut Kandemir, Yuan Xie. Temperature-Sensitive Loop Parallelization for Chip Multiprocessors, in the *proceedings of ICCD 2005.*