Performance Aware Secure Code Partitioning *

S. H. K. Narayanan and M. Kandemir
The Pennsylvania State University
{snarayan,kandemir}@cse.psu.edu

R. Brooks
Clemson University
rrb@acm.org

Abstract

Many embedded applications exist where decisions are made using sensitive information. A critical issue in such applications is to ensure that data is accessed only by authorized computing entities. In many scenarios, these entities do not rely on each other, yet they need to work on a secure application in parallel to complete application execution under the specified deadline. Our focus in this paper is on compiler-guided 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 various hosts have different levels of access to the data structures manipulated in the application. Our approach partitions the application among the hosts such that the load imbalance across hosts is minimized to reduce execution time while ensuring that no security leak occurs.

1. Introduction

Many embedded as well as distributed applications exist where decisions are made using sensitive information [6, 9, 18]. The execution environment of such applications is among hosts that have varying degrees of access to data such as the one outlined in [2]. In such an environment, the application is partitioned so as to take into account the different inter-host trust levels and data sharing patterns [18, 20]. Our focus, in this paper, is on performance aware compiler-guided 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 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 [2, 15]. 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 [14] but has wide practical usage [8].

A classic MLS lattice is shown in Figure 1(a) [2]. 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 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 [13]. An example of host subjects which are assigned different clearance levels is shown in Figure 1(c). Data objects assigned various sensitivity levels are shown in Figure 1(d). A host is allowed to access a particular data if the following two conditions are met [2, 15]:

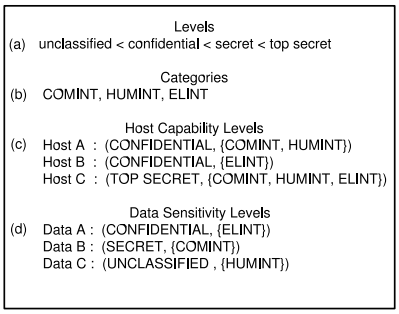

Figure 1: Multilevel security. (a) Lattice of different security levels (b) Categories of data (c) Host capability levels of (d) Data sensitivity levels.

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 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 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 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 paper 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) [17]; in this paper 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.

The rest of this paper is structured as follows. The next section discusses our compiler-driven approach to secure code partitioning in detail. Section 3 presents an experimental evaluation using two representative application scenarios. Section 4 discusses related work, and Section 5 summarizes the paper.

*This work is supported in part by NSF Career Award 0093082 and by a grant from the GSRC.

978-3-9810801-2-4/DATE07 © 2007 EDAA

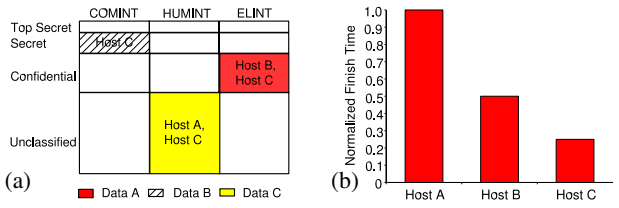

Figure 2: (a) Data accessible to the different hosts. (b) Normalized execution time for the hosts without load balancing.

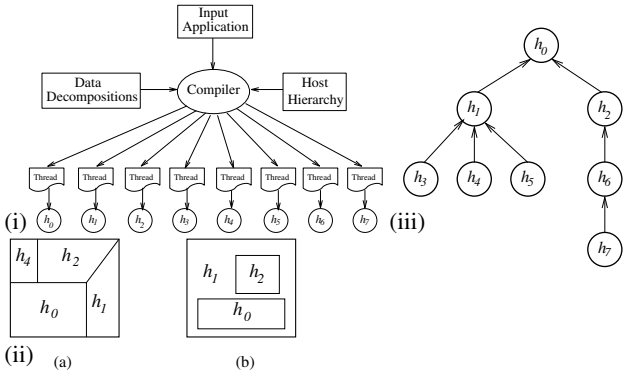

Figure 3: (i) Secure code partitioning. (ii) Two different data decompositions. (iii) An example host hierarchy tree (HHT).

2. Code Partitioning

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 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 can have a different type of decomposition and can be accessed by (potentially) different sets of hosts. For example, Figure 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 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 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.

Table 1: Notations used in the paper.

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

2.2 Representation of Data and Iteration Sets

To identify loop iterations that access data in a particular array region, we use Presburger formulas [10]. Presburger formulas are those formulas that can be constructed by combining affine constraints on integer variables with the logical 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 1 gives some of the notation used in this paper; 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 $I(h_j, I_k)$.

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 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 paper, 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,

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

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 [16]. In the rest of the paper, $|I_{init}(h_i, I_k)|$ 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 $I_{init}(h_j, I_k) \subseteq I(h_j, I_k)$.

2.4 Workload Balancing Algorithm

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 2.3) and operates on the HHT under consideration. Its goal is to redistribute 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 preorder 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.

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 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 1 $ReassignHHT()$

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**

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 paper, 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 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, $carryout(h_{child}, I_k)$

Algorithm 2 $BottomToTop(h_i, N_{avg})$

```
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:     |Γ(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:     |Γ(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 - |Γ(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:            |Γ(h_i, I_k)| := |Γ(h_i, I_k)| + carryout(h_child, I_k)
17:            carryout(h_child, I_k) := 0
18:          else
19:            |Γ(h_i, I_k)| := |Γ(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
```

Algorithm 3 $TopToBottom(h_i, N_{avg}, carry\_reduce)$

```
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)|-|Γ(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-|Γ(h_i, I_k)| > carryout(h_i, I_k) then
12:      |Γ(h_i, I_k)| := |Γ(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-|Γ(h_i, I_k)|
17:      adjust reduced carryout recursively to h_j s.t. h_i ▷ h_j
18:      tot_red:= N_avg-|Γ(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
```

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.

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 the value $carryout(h_i, I_k)$, 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.

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 [3], shown below written in a pseudo language. The HHT considered for this example is shown in Figure 4(a). The data space decompositions for arrays A and B given in Figure 4(b), and Figure 4(c) shows the initial workloads for the hosts. Figure 4(d) shows the snapshot of the HHT after the BottomToTop procedure operates. As the tree is unbalanced, procedure TopToBottom is called. Figure 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

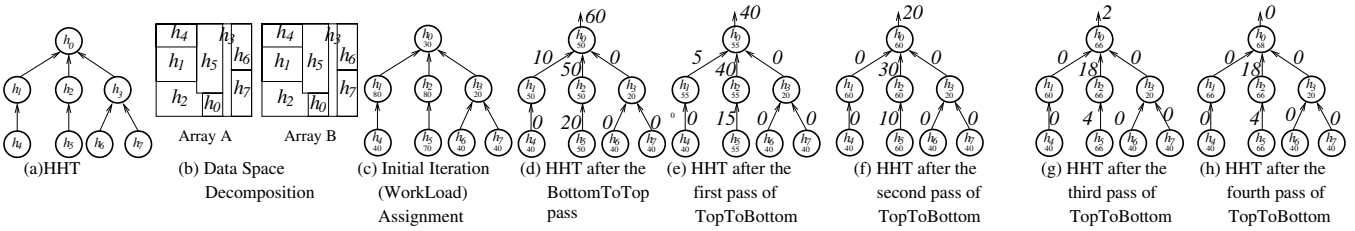

Figure 4: (a) An example HHT. (b) Example data decompositions. (c) Initial workload assignment. (d) Situation after BottomToTop. (e-h) Situation after different passes of TopToBottom. The numbers within the nodes denote the current loads (in terms of loop iterations) and the numbers along the arrows indicate carryouts.

passes of the procedure are shown in Figures 4(f), (g), and (h).

```
for(i = 2  to  N − 1)
 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
```

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 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 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:

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

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:

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

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 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.

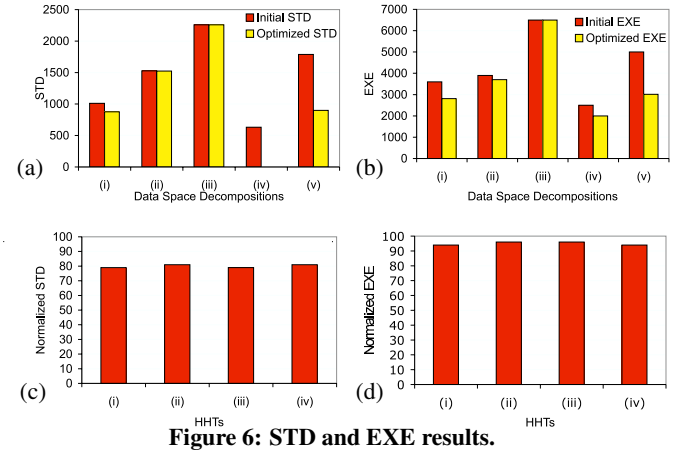

Figure 6: STD and EXE results.

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

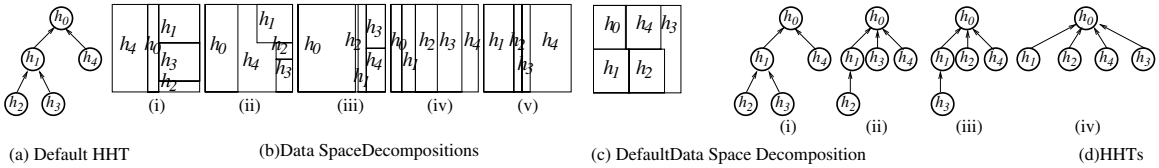

Figure 5: Setup for the first experimental scenario: (a) and (d) HHTs. (b) and (c) data decompositions.

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(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 6(a) and (b) give, respectively, the STD and EXE metrics for this scenario under the data decompositions shown in Figure 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(c) and use different HHTs shown in Figure 5(d). The experimental results are given in Figures 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.

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 [4]. Jif [7] and CQUAL [1] are examples of current tools that perform static analysis of a program through analysis of the security types present in the program. SELinux [12] is an operating system that provides support for security aware applications. Other distributed operating systems that have been proposed such as [5] which execute code across multiple hosts keeping performance in mind. Load balancing across different hosts in a distributed environments is a well researched topic [3]. In contrast to the distributed environment, our environment does not consider communication costs. [19] 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 [11]. Maybe the most relevant prior work to ours is [18, 20], 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. 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 paper 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.

6. References

[1] http://www.cs.umd.edu/ jfoster/cqual/

[2] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. *MITRE Technical Report 2457*, Volume 1, 1973.

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

[4] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM,* 20(7):504–513, July 1977.

[5] F. Douglis, J. K. Ousterhoutm, M. F. Kaashoek, and A. S. Tannenbaum. A comparison of two distributed systems: Amoeba and Sprite. *ACM Transactions on Computer Systems*, 4(4), Fall 1991.

[6] I. Foster, N. T. Karonis, C. Kesselman and S. Tuecke. Managing Security in High Performance Distributed Computations. In *Cluster Computing*, Vol 1, issue 1, pages 95-107, 1998.

[7] http://www.cs.cornell.edu/jif/

[8] P. A. Karger. Multi-Level Security Requirements for Hypervisors. In *Proceedings of ACSAC*, 2005.

[9] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi Security as a New Dimension in Embedded System Design. In *Proceedings of DAC*, 2004.

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

[11] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of ESEC/FSE,* 2003.

[12] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *the Proceedings USENIX '01.*, 2001.

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

[14] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology,* 9(4):410–442, 2000.

[15] P. G. Neumann, R. J. Feiertag, K. N. Levitt, and L. Robinson. Software development and proofs of multi-level security. in *the Proceedings of ICSE,* 1976.

[16] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proceedings of PLDI,* pp. 121–134, 1994.

[17] K. Thulasiraman and M.N.S. Swamy. Graphs : theory and algorithms, *Wiley*, c1992.

[18] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *Proceedings of SOSP,* 2001.

[19] X. Zhang and R. Gupta. Hiding program slices for software security. In *proceedings of CGO,* 2003.

[20] L. Zheng, S. Chong, S. Zdancewic, and A. C. Myers. Building secure distributed systems using replication and partitioning. In *Proceedings of the IEEE Symposium on Security and Privacy.*, 2003.