Memory Binding for Performance Optimization of Control-flow Intensive Behaviors

Kamal S. Khouri[†], Ganesh Lakshminarayana[‡], and Niraj K. Jha[†]

[†]Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

[‡]C&C Research Labs
NEC USA, Inc., Princeton, NJ 08536

Abstract

This paper presents a memory binding algorithm for behaviors that are characterized by the presence of conditionals and deeply-nested loops that access memory extensively through arrays. Unlike previous works, this algorithm examines the effects of branch probabilities and allocation constraints. First, we demonstrate, through examples, the importance of incorporating branch probabilities and allocation constraint information when searching for a performance-efficient memory binding. We also show the interdependence of these two factors and how varying one without considering the other may greatly affect the performance of the behavior. Second, we introduce a memory binding algorithm that has the ability to examine numerous bindings by employing an efficient performance estimation procedure. The estimation procedure exploits locality of execution, which is an inherent characteristic of target behaviors. This enables the performance estimation technique to look at the global impact of the different bindings, given the allocation constraints.

We tested our algorithm using a number of benchmarks from the parallel computing domain. A series of experiments demonstrates the algorithm's ability to produce bindings that optimize performance, meet memory allocation constraints, and adapt to different resource constraints and branch probabilities. Results show that the algorithm requires 37% fewer memories with a performance loss of only 0.3% when compared to a parallel memory architecture. When compared to the best of a series of random memory bindings, the algorithm improves schedule performance by 21%.

1 Introduction

Today's embedded applications typically work with large data sets which are stored in memories. The memory accesses performed by an application are represented as array accesses in the behavioral specification. The manner in which arrays in the behavior are mapped to physical memory (memory binding) is an important determinant of the performance of the design. In other words, the quality of the generated schedule depends upon the binding information assumed by the scheduler. This paper focuses on performing memory binding to preserve the parallelism in the application so that a high-performance schedule can be derived for it. Our technique accepts as inputs, a control-flow intensive (CFI) behavior, which is a functional specification with significant control-flow in the form of nested loops (possibly with unknown iteration bounds) and conditionals, memory allocation information (information about the numbers and attributes of memories available for the design), and functional unit allocation information (numbers and types of functional units available), and returns a mapping of arrays in the functional specification to the allocated memories. The mapping is performed with the objective of optimizing the performance of the scheduled behavior.

The key features of our technique are as follows:

- It is well-known that mapping arrays to physical memory introduces extra dependencies in the behavior, which in turn imposes constraints on the schedule. For example, if two arrays, which belong to loops that are independent of each other, are mapped to the same single-ported physical memory, then the loops cannot execute in parallel, and must be sequentialized. Our analysis, which is based on a global view of the functionality, understands and exploits these dependencies to find a performance-efficient memory binding.
- We demonstrate that branch probabilities and functional unit allocation constraints can have a significant impact on the efficiency of memory binding. Unlike past work in the area, our algorithm takes these factors into account.
- The ability of the algorithm to determine an efficient memory binding is determined, to a large extent, by the efficacy of the performance estimation procedure which evaluates the binding. This procedure needs to determine the impact of diverse factors, such as the structure of the loop hierarchy, functional unit allocation constraints, and branch probabilities, on the performance of the design. Since many candidate bindings need to be evaluated, a performance estimation procedure which performs a complete schedule, taking all these factors into account, would be computationally infeasible. This is especially true of complex and aggressive schedulers which perform several optimizations like pipelining, loop unrolling, and concurrent loop optimization. We address this issue by using a fast, high-level performance estimation technique that we have devised. It estimates the performance of the scheduled behavior without actually performing scheduling.

The rest of the paper is organized as follows. Section 2 discusses past work and situates our work in the appropriate context. Section 3 illustrates key algorithmic ideas through examples. Section 4 describes our algorithm. Section 5 reports experimental results and Section 6 concludes.

2 Related Work

Memory organization and optimization have been the focus of extensive research in the context of general-purpose processors. Memory hierarchy design, cache organization, size, and replacement policies are among the well-studied problems [1]. Since memory is large and monolithic, critical problems in this domain include reducing memory access times, carefully designing the memory hierarchy, and organizing data to reduce cache miss rates and access times.

Compact storage is essential for application-specific integrated circuits (ASICs), and memory organization techniques targeted towards reducing the area of memory have been proposed. Techniques to estimate the memory required to implement a functional specification have been proposed in [2, 3]. Transformations to reorganize the loops and conditionals in the behavior to reduce the required memory have been presented in [4].

Past work in storage binding has considered the mapping of scalar variables to registers, register files and small memories [5, 6]. Some high-level synthesis systems map arrays in the behavior into a single, monolithic memory [7, 8, 9, 10]. This approach has the advantage of being similar in concept to general-purpose processors (which have a single memory hierarchy), and can, therefore, use features like pointers and dynamic memory allocation. However, this approach is too restrictive for ASIC designs. On the other end of the spectrum, techniques that map each array to a separate memory have been proposed [11]. Many optimization and exploration techniques for memories in embedded systems have been addressed in [10]. These include exposing the off-chip memory architecture at the behavior level by introducing memory-architecture specific access nodes in the behavioral description. Another contribution of this work is in minimizing the off-chip memory access penalty. This is achieved by exploring different memory organizations and utilizing *scratch-pad memory*.

For the remainder of this section, we focus on techniques which handle heterogeneous memory architectures, and allow multiple arrays to share physical memory.

A technique to reduce memory size, by mapping multiple arrays into the same physical memory, has been presented in [12]. Its objective is to reduce memory size to the greatest extent possible, under performance constraints. It constrains different arrays to occupy different ranges of memory locations, within the same physical memory. Therefore, different arrays cannot share the same address space. It was primarily concerned with avoiding access conflicts while scheduling and employed a heuristic based on reducing the maximum number of simultaneous read/write accesses. It has been improved to include more general kinds of array mappings in [13], where different arrays are allowed to share the same address space, and still occupy disjoint regions of memory. For example, an array of 12-bit words and an array of 8-bit words are allowed to share address space in a 20-bit memory. Also, the problem formulation permits memory area optimization under performance constraints, and performance optimization under constraints on the available memories. The technique presented in [14] guides memory binding by identifying groups of arrays that can potentially occupy the same memory locations. It returns an *extended conflict graph* whose nodes represent arrays and whose edges represent compatible groups of arrays (those that can be mapped to the same physical memory). The edges are annotated with the numbers of read/write ports needed by a memory which stores the arrays representing the nodes they connect. The method presented in [15] uses *polyhedral analysis* in an integer-linear program (ILP) formulation to find the optimal memory architecture for a given performance constraint. Most of the above approaches do not analyze the effect of allowing different arrays to share the *same address space*. While some techniques allow this sharing [15], its effect on performance is not modeled and accounted for. Sharing is an important means of reducing memory size, but must be judiciously employed in order to ensure high-performance designs, due to the extra dependencies it introduces. Past work also ignores the influence of functional unit allocation and branch probabilities on memory binding. We demonstrate that these factors play an important role in memory binding, and are tightly coupled with each other. We follow up this observation with an estimation technique that distills these diverse factors into a single performance estimate, which we use for evaluating candidate bindings. Our performance estimation technique is driven by the locality of execution rule [1], which exploits the fact that the performance of a CFI design is frequently determined by a few critical statements of code, which dominate the computational effort. Identification and analysis of these critical segments enables us to perform a fast estimate of the quality of a given binding, without actually having to perform scheduling. As a result, our technique can search through a large number of candidate bindings within a short time to converge on an efficient solution.

3 Motivation

In this section, we motivate our key ideas through a number of simple examples. Example 1 illustrates the impact that memory sharing has on the performance of the design. This example motivates judicious selection of arrays to share address space. Example 2 brings out the impact of branch probabilities. It demonstrates that a change in branch probabilities can cause the best binding to change, when all others factors are kept constant. Example 3 shows that constraints on functional unit allocation can enhance or decrease the criticality of individual code segments. This, in turn, influences memory binding. For the purpose of illustrating the above points, these examples contain only read accesses to memory. However, our algorithm can handle multiple (from multi-port memories) and more general reads and writes just as well.

```
int Test1 (Array<int> x1, x2, x3, x4, x5, x6) {
  int v, y, y′, z;
  v = y = y′ = z = 0;
  for(int i1 = 0; i1 < 4000; i1++) { // <1, ++1 (L1)
    y′ = x1[i1] − y′; //−1
  }
  for(int i2 = 0; i2 < 4000; i2++) { // <2, ++2 (L2)
    y = x2[i2] + x2[i2 − 1]; //+1, −2
    z = z + y; //+2
    y = y + y; //+3
  }
  for(int i3 = 0; i3 < 3000; i3++){ //<3, ++3 (L3)
    if(x3[i3] > 27){//>1
      v = v + x3[i3];//+4
    }
    else {
      v = v << 2; //<<1
      v++; //++4
      v = v >> 4; //>>1
      v = v × 3; //∗1
      v = v × x3[i3]; //∗2
      v−−; //− − 1
  }
  for(int i4 = 0; i4 < 4000; i4++) { //<4, ++5 (L4)
    y′ = y′ − x4[i4] //−3
    z = z − y′; //−4
  }
  for(int i5 = 0; i5 < 250; i5++) { //<5, ++6 (L5)
    v = v + x5[i5]; //+5
    v = v − z; //−5
  }
  return v;
}
```

Figure 1: A fragment of code, called *Test1*, written in a high-level language

Example 1: Consider the behavioral description, *Test1*, shown in Figure 1. Loops in the behavior are annotated with their names. *Test1* consists of five loops, *L1*, *L2*, *L3*, *L4*, and *L5*. Statements in the behavior are annotated with the names of the operations they represent. For example, operation -1 represents statement $y' = x1[i1] - y';$. *Test1* has five arrays $x1$, $x2$, $x3$, $x4$, and $x5$, of lengths 4000, 4000, 3000, 4000, and 250, respectively. All arrays are assumed to have a width of 4 bytes (32 bits). The functional unit allocation is as follows: three incrementers of type *inc1*, three comparators of type *cmp1*, one left shifter of type *lsh1*, one right shifter of type *rsh1*, two subtractors of type *sub1*, four adders of type *add1*, and one multiplier of type *mult1*. For the purpose of this example, all functional units are assumed to take one cycle to execute. Consider the problem of mapping these arrays

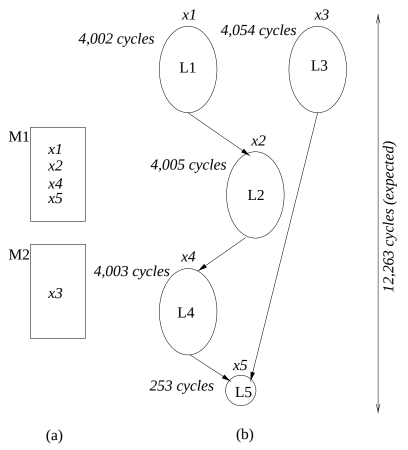

Figure 2: (a) Memory binding *A*, (b) schedule of *A*

to two memory modules, *M1* and *M2*, each of capacity $4K \times 32bits$. To illustrate the performance implications of memory binding, we consider two different bindings, *A* and *B*. Binding *A* (*B*) is described in

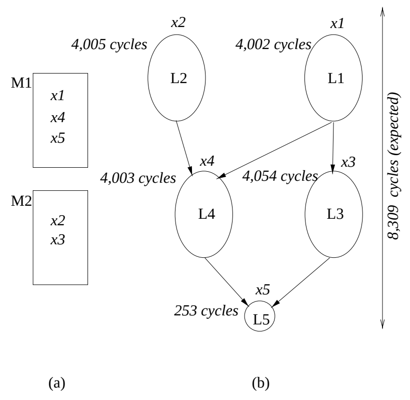

Figure 3: (a) Memory binding *B*, (b) schedule of *B*

Figure 2(a) (Figure 3(a)). Since binding *A* requires arrays *x1* and *x2* to share the same physical space, loops *L1* and *L2* cannot execute in parallel. Therefore, the execution of these loops needs to be sequentialized in the schedule. Figure 2(b) describes, at a high level, a schedule produced from binding *A*. The nodes in Figure 2(b) represent loops, and are annotated with their execution times. For example, loop *L1* can be shown to take 4,002 cycles to complete execution. Loops *L1* and *L3* execute in parallel because the memory binding chosen does not introduce a dependency between them. The execution time for loop *L3* was derived assuming that operation >1 evaluates to *true* with a probability of 0.93. The schedule completes, on an average, in 12,263 $cycles$. The critical path, in this schedule consists, in part, of loops *L1* and *L2*, executing in sequence. We now consider binding *B*, which attempts to reduce the length of the critical path by breaking the dependency between loops *L1* and *L2*. This is done by mapping *x1*, *x4*, and *x5* to one memory module, and *x2* and *x3* to another. This breaks the dependency between loops *L1* and *L2*, but sequentializes loops *L2* and *L3*. We are, however, free to determine which of *L2* and *L3* evaluates first. Since *L2* is on a critical path, we execute *L2* before *L3*. The resultant schedule is shown in Figure 3(b). As we can see, the schedule completes, on an average, in 8,309 $cycles$, which is significantly smaller than the 12,263 $cycles$ consumed by binding *A*. Our algorithm, presented in Section 4, works with a high-level view of the schedule, like the one shown in Figure 3(b), and clearly analyzes and accounts for the constraints memory binding imposes on the loop structure of the behavior. ∎

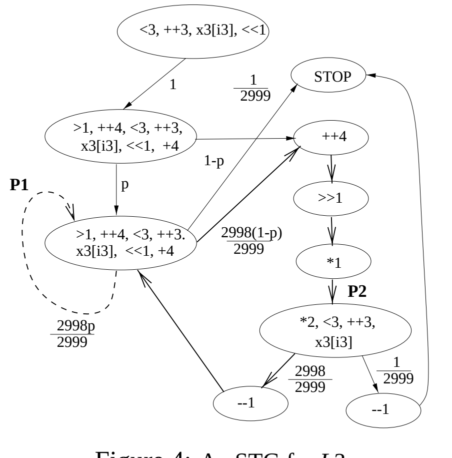

Figure 4: An STG for *L3*

Example 2: Consider again the behavior shown in Figure 1. We consider the problem of mapping the arrays in *Test1* to two $4K \times 32bit$ memory modules, *M1* and *M2*, as before. The functional unit allocation constraints are the same as those used in Example 1. However, the probability p that operation >1 evaluates to *true* is now assumed be 0.4, instead of 0.93. Figure 4 shows the state transition graph (STG) which represents a schedule for loop *L3* [1]. The STG is derived using aggressive scheduling techniques such as loop-unrolling and speculative execution. Paths $P1$ (shown in dashed lines) and $P2$ (shown in bold) are the main determinants of the time taken for loop *L3* to complete. The expected execution time, t_{L3}, of *L3* can be approximated by the equation:

$$t_{L3} = 3000 \times (p \times |P1| + (1-p) \times |P2|)$$

where $|P1|$ denotes the length of $P1$ and $|P2|$ is the length of $P2$. Substituting $|P1| = 1$ and $|P2| = 6$, t_{L3} can be found to change to 12,004 $cycles$ (4 *cycles* are required for setup) from its initial value of 4,054 $cycles$. Figure 5 presents the schedule derived, assuming bind-

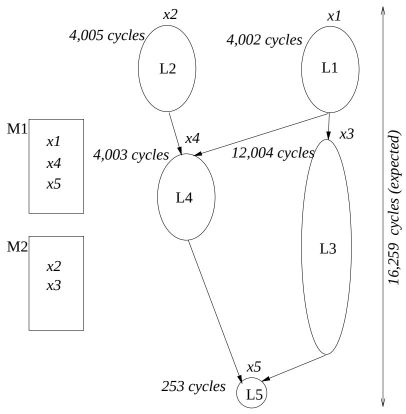

Figure 5: Performance of *Test1* with binding *B*

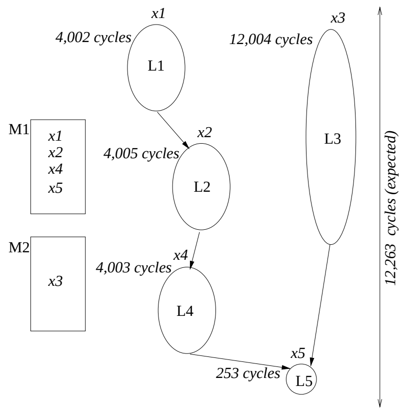

Figure 6: Performance of *Test1* with binding *A*

ing *B*, under the new branch probabilities. The schedule takes, on an average, 16,259 $cycles$ to complete. The new critical path in the design consists of the sequence of loops *L1*, *L3*, *L5*. In this case, it can be seen that better performance can be obtained by reverting to binding *A*, as shown in Figure 6. Binding *A* causes the expected schedule length for *Test1* to decrease to 12,263 $cycles$ from 16,259 $cycles$, an improvement of 24.6%. Clearly, identifying the critical path in the behavior, under a given memory binding, requires detailed evaluation of the performance of individual loops in the design, and considering probabilities of individual branches. Note that in this example, it was possible to evaluate the performance of a loop *in isolation*. For instance, the time taken for loop *L3* to execute was 12,004 $cycles$ under bindings *A* and *B*. This is because the parallel loops in these examples do not contend for resources [2]. In general, this may not be the case. If multiple parallel loops contend for resources, the time taken by a loop to execute

[1]Nodes in the STG represent states and edges represent transitions between states. Edges in the STG are annotated with their probabilities, and states are annotated with the operations performed in them.

[2]The resource allocation ensures that all loops can execute at their maximum rate, independent of each other.

depends on its environment, *i.e.,* the loops it is scheduled in parallel with. For example, if the number of adders allowed were reduced to three, from four, *L3* and *L2* would contend for resources under binding *A*. This is because *L2* requires three adders to begin an iteration every cycle, and *L3* requires an adder every cycle if >1 evaluates to *true*. This implies a resource requirement of four adders per cycle, instead of the three supplied. Thus, in general, the execution time of individual loops cannot be evaluated independent of each other. The performance analysis procedure will have to take into account the parallel execution of loops, and the resultant resource contention introduced. ∎

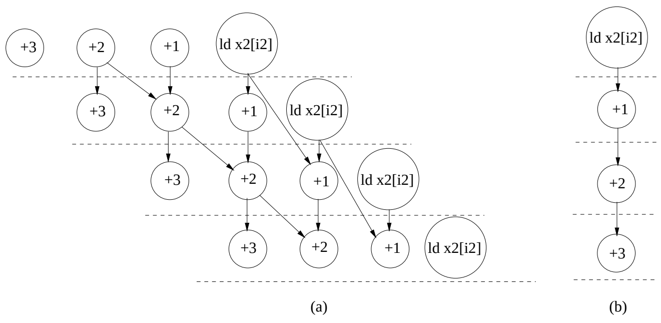

Figure 7: Execution of loop *L2* under (a) resoure constraint of Example 1, and (b) new resource constraints

Example 3: In this example, we consider the problem of performing memory binding for *Test1* under the following functional unit allocation: three incrementers of type *inc1*, three comparators of type *cmp1*, one left shifter of type *lsh1*, one right shifter of type *rsh1*, two subtractors of type *sub1*, one adder of type *add1*, and one multiplier of type *mult1*. All functional units are assumed to take one cycle to execute. Branch probabilities are the same as those used in Example 2. We first consider memory binding *A*: $\{x1, x2, x4, x5\}$ which maps to memory module *M1*, and $\{x3\}$ which maps to memory module *M2*. Figure 7

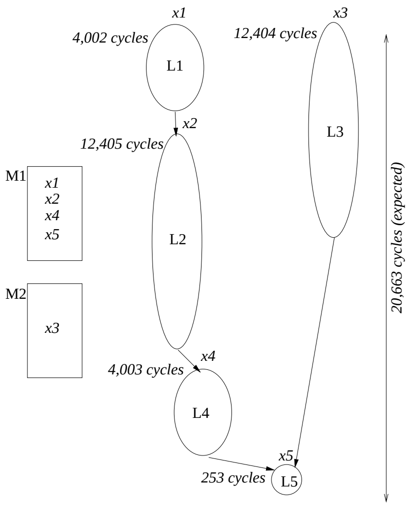

Figure 8: Memory binding *A* for *Test1* in Example 3

outlines the schedule for loop *L2* under the allocation of Example 1 (where four adders were available) and new resource constraints (one adder only). Four adders enabled *L2* to start a new iteration every cycle, as shown in Figure 7(a). The reduction in the number of adders from four to one limits the initiation time (the interval, in cycles, between consecutive loop iterations) of *L2* to three cycles (the loading of $x2[i2]$ needed for the next iteration can be done in parallel with an addition operation), as shown in Figure 7(b). Therefore, the time taken by *L2* to execute increases by a factor of three. Also note that, under memory binding *A*, part of *L2* executes in parallel with *L3*. Since *L3* requires an adder when operation >1 evaluates to *true*, *L3* and *L2* contend for the use of the adder. To sustain its initiation rate of one iteration every three cycles, *L2* needs to use the adder every cycle (see Figure 7(b)).

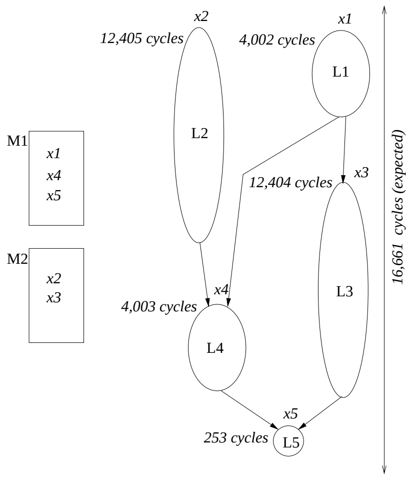

Figure 9: Memory binding *B* for *Test1* in Example 3

Therefore, the resource contention further increases the times taken by loops *L2* and *L3*. Figure 8 outlines the schedule under memory binding *A*. When *L2* begins to execute, approximately 1,000 iterations of *L3* will have completed execution. Of the remaining 2,000 iterations 800 are expected to be the true branch of >1; *i.e.,* there are 800 additions that will conflict for resources with *L2*. If we assume the conflicts are resolved such that each loop receives an equal share of the extra cycles, the execution times for loops *L2* (*L3*) increases to 12,405 $cycles$ (12,404 $cycles$).

Clearly, an increase or decrease in the number of resources can:

- increase/decrease execution time of individual loops.
- introduce/resolve resource conflicts between loops.

If we choose memory binding *B*, the critical path changes to consist of the sequence of loops *L2*, *L4*, *L5*. The critical path is thus significantly reduced by moving $x2$ from *M1* to *M2*, as shown in Figure 9. Under this binding, the schedule takes only 16,661 $cycles$ to complete, as opposed to 20,663 in binding *A*. Section 4 presents techniques to take these effects into consideration while evaluating a candidate memory binding. ∎

4 The Memory Binding Algorithm

In this section, we present the details of our memory binding algorithm. Figure 10 is a block diagram of the algorithm which is composed of two blocks: the binding heuristic and performance estimation algorithm. The sections to follow are a detailed description of these blocks and their importance. The input to the algorithm is a control-data flow graph (CDFG) of the behavior and a resource constraint set which contains the number and types of the different functional units, registers and memories to be used.

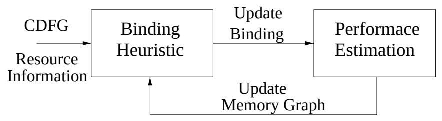

Figure 10: Our memory binding algorithm

Section 4.1 explains our performance estimation algorithm, Section 4.2 describes the graph model used to capture different memory bindings, and Section 4.3 combines these two concepts which are used to produce a memory binding that attempts to maximize performance and meet the allocation constraints.

4.1 Performance Estimation

As illustrated in the examples of Section 3, there are many factors that interact when performing memory binding for CFI behavioral descriptions: branch probabilities, allocation constraints, inter- and intra-loop dependencies. Ultimately, these interactions affect the performance of the schedule. In order to examine many candidate bindings, without having to evaluate the full impact of each factor, we need an accurate, yet fast, means of estimating the performance impact of each candidate.

Evaluation of a given schedule may be categorized into one of two methods:

1. Simulation-based estimation: In this method, the STG is simulated using a typical input trace provided by the designer or end-user. The performance is a count of the number of cycles that are required to complete the simulation using the input stimuli. While this method is very accurate in determining the performance for the given trace, it is computationally intensive and infeasible when required to evaluate a large number of schedules.

2. Markov model estimation: In this method, the STG is modeled as a Markov chain and the performance evaluation is based on the information on the state-transition probabilities and state probabilities [16]. The computational complexity of this method is $O(n^3)$, where n is the number of states in the STG. While this method is faster than method (1), it is still not fast enough since an average size CFI schedule[3] can easily have over 200 states when an aggressive scheduling technique (one that performs loop unrolling, parallel loop optimization and speculative execution) is used.

Note that the above two techniques depend on a full and functionally correct schedule. Our method takes advantage of two observations that allow us to produce a relatively accurate estimate of the performance without consuming too much CPU time. The first observation is that, for the sake of a performance estimate, it is not necessary to have a schedule that emulates the exact functionality of the design, but one that emulates the performance. The second observation lies in the nature of CFI designs. Such designs typically exhibit the *90/10 locality rule* [1]. This rule states that 90% of a program's execution time is spent in 10% of its code. In other words, the behavior is dominated by a set of critical paths which constitute a small portion of the STG. These paths need to be identified and their execution time needs to be estimated. For this purpose, the only requirement is to generate and examine the high probability threads of execution.

The CDFG may be viewed as a hierarchy of nested loops[4], and this hierarchy can be used to induce a partition on the STG. To estimate the performance of a schedule, we need to identify which partitions are highly probable, find the length of each partition and then combine the lengths appropriately. We now formalize this procedure using a simple example and with the help of two abstractions: *phase graph* and *cycle processor*, which will aid in visualizing the estimation process. At this point, we assume that a binding is given to the performance estimator (from the binding heuristic), as shown in Figure 10.

4.1.1 The Phase Graph

A phase graph, shown in Figure 11, is an acyclic graph whose nodes represent phases and edges represent transitions from one phase to other phases. A *phase*, Φ_S, is defined as a set of loops, S ={*L1*, *L2*, ..., *LN*}, executing simultaneously. Each edge, $\Phi \rightarrow \Phi'$, in the phase graph is associated with a transition probability, $p_{\Phi\rightarrow\Phi'}$.When any loop changes state, *i.e.,* falls through, or a new loop begins to execute, a new phase, $\Phi_{S'}$, is entered where a new set, S', of loops is executed. The complete phase graph can be easily extracted from the CDFG and a given memory binding. As mentioned above, we can partition the CDFG into regions defined by the loops which are part of the behavior. The partitioning identifies a partial ordering of loops, *i.e.,* which loops are dependent and, hence, cannot be executed simultaneously. The binding information introduces additional dependencies and further orders the loops.

Since we are only interested in the high probability threads, we simply prune out all improbable paths. The probability of a path in the phase graph is the product of the probabilities of the edges that are on the path and, hence, we define a threshold probability, p_{th}, with

[3]Unlike data-dominated designs, which have a fixed schedule length (say n cycles per iteration), a CFI schedule has multiple threads of execution. The thread that is executed during any given iteration depends on the values of the primary inputs.

[4]The CDFG is itself a loop since it executes repeatedly under a given set of inputs.

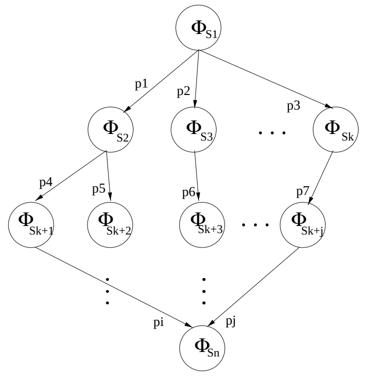

Figure 11: A generic phase graph

which we prune low probability edges. A smaller p_{th} means more paths to consider and, hence, more CPU time is consumed evaluating these paths. The value of p_{th} is set by the end user who may have some insight about the memory usage of the behavior, or simply as a means to experiment with different levels of threshold. The performance estimate is then given as the sum of execution times over all phases. In other words, the expected length of a phase graph is:

$$T_{estimate} = \sum_{\forall\ sets\ S} T(\Phi_S) \times p(\Phi_S) \qquad (1)$$

where $T(\Phi_S)$ is the time taken for phase Φ_S to execute and $p(\Phi_S)$ is the probability of entering phase Φ_S. In the next section, we describe how the algorithm estimates the time taken to execute each phase for a given binding and resource constraint set.

4.1.2 Phase Time Calculation

In this section, we describe how to evaluate the time taken to execute each phase. To do this, we use the cycle processor abstraction. The primary goal of this abstraction is to estimate the time taken to execute various loops in the behavior as if an STG had been given.

The inputs to the processor are the resource constraint set and the portion of the behavior that is to be considered. The processor may be viewed as a machine with a set of resources on which the behavior is running as a program. While a scheduler will use the resource set to assign operations to states, the cycle processor only performs a logistic task and simply counts the occurrence of operations that execute within the various loops in the behavior. The processor proceeds to assign hardware to operations and evaluates the rate (given in operations per cycle) of each operation. This is illustrated in Figure 12.

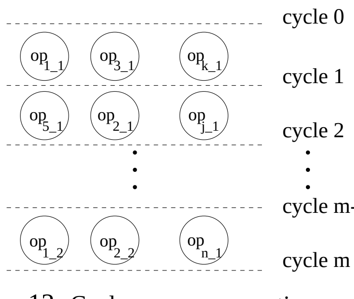

Figure 12: Cycle processor counting operations

Consider a loop L, with operations $(op_1, op_2, \cdots, op_n)$, and assume that these operations are unconditional; *i.e.,* they all execute with a probability 1. In Figure 12, op_{j_i} represents operation op_j which is part of iteration i. The rate of operation op_j, denoted by $r(op_j)$, is the number of times op_j executes per cycle. We define an iteration i of L to be complete if and only if all the operations that belong to i have completed execution. For example, in Figure 12 it takes m cycles for iteration 1 to complete since op_{n_1} is accounted for in the m^{th} cycle. Note that op_1 has been accounted for twice in these m cycles. With this in mind, if we consider a large number of cycles, N, then the number of times the loop has completed execution is less than or equal to the number of times an operation, op_k, has been counted. Due to the imposed resource constraints, we can find an operation, op_n, which has executed a minimum number of times. Since all other operations have completed at least that number of times, the rate of the operation dictates the rate of the loop. In other words, the number of times the slowest operation, op_n, has appeared is equal to the number of times the loop has executed. This means that the rate, r_L, of loop L is:

$$r_L = MIN_{\forall i}\left\{r(op_i)\right\} \tag{2}$$

It is the case with CFI behaviors that many operations do not execute with probability 1 due to branching and nested loops. Consider the same loop L, and an operation, op_i, which has a probability, p_i, of execution during a cycle. Using Chebyshev's inequality, we can show that for a large number of cycles, N, with a high probability the following inequality holds for n, the number of times a loop has executed:

$$n \leq \frac{r(op_i)}{p_i} \tag{3}$$

We can also show here that the rate at which a loop executes is determined by its slowest operation and, hence, the rate, r_L, of loop L is given by:

$$r_L = MIN_{\forall i}\left\{\frac{r(op_i)}{p_i}\right\} \tag{4}$$

In turn, the rate of a phase, Φ, is determined by the slowest loop and hence we can replace r_L in Equation (4) by r_Φ.

The processor continues to evaluate operation rates until all rates converge to a steady value. Convergence occurs when the rate of an operation is the same for two consecutive iterations of the loop the operation belongs to. Given the rate of the slowest operation and the total number of cycles, N, the processor ran for, we can compute the number of times the loops in the phase executed as the product of r_Φ and N.

4.2 Memory Binding Graph Model

In this section, we describe our memory binding graph model and a heuristic to select candidates for binding. The graph is non-directed. It is composed of nodes which represent arrays and edges connecting nodes which can potentially share the same memory module. The edges are inferred from the data and control dependencies among the arrays, and are extracted from the CDFG. Each edge is annotated with a weight, w, that represents the cost (given in terms of performance) if two connected nodes share the same memory module. A solid edge means that the nodes can share the address space, a dashed edge means that they can share the same memory module but not the address space, and no edge (or edge with $w = \infty$) means the nodes cannot share the same memory module.

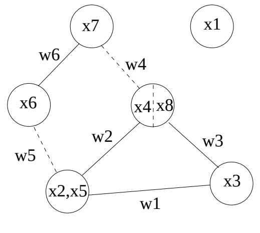

Figure 13: An example binding graph

If the behavior consists of m arrays, then the initial graph will have m nodes, such that each array is assumed to occupy a separate memory module. When two arrays are identified as candidates for sharing, they are collapsed into a single node. The candidates for collapsing may be chosen by means of any search algorithm that uses the edge weights as a search criterion. The collapsing and search process continues until either the required number of memory modules is attained or when the performance of the behavior is at the desired level. This approach has the added flexibility of allowing the designer to provide a memory module constraint or a performance constraint.

Figure 13 illustrates an example of an intermediate binding graph of eight arrays. This example shows that array $x1$ cannot share the same memory with other arrays, that arrays $x2$ and $x5$ share the same memory module and address space, and that arrays $x4$ and $x8$ share the same memory module. Two nodes n_i and n_j are connected if there does not exist a dependency in the CDFG between them or their combined size is less than that of a given memory module. The relation is denoted by the notation $\epsilon(n_i, n_j)$. If nodes n_i and n_j are collapsed into a new node n_k, then an edge exists between n_k and a node n_l, denoted by $\epsilon(n_k, n_l)$, only if $\epsilon(n_i, n_l)$ AND $\epsilon(n_j, n_l)$ is true.

In the next section, we show how to use the graph model and the performance estimator to perform memory binding.

4.3 Putting it all together

Figure 14 is the pseudo-code for the binding algorithm. The input to the algorithm is the CDFG, memory and functional unit allocation

```
BIND_MEMORY (CDFG $C$, Allocation ($M$, $F$), Probabilities $P$) {
  $G$ = Graph($V, E$);
  for_each_array ($a$ in $C$) {
    Node $n$ = ADD_NODE($a$, $G$) {
      for_each_node ($n'$ in $G$) {
        if (!dependent(GET_ARRAY($n'$), $a$) AND !edge($n'$, $n$)) {
          Edge $e$ = ADD_EDGE($n$, $n'$, $G$);
          cost($e$) = 0; }
        else if (size(GET_ARRAY($n'$), $a$) $\leq$ GET_SIZE($M$, $a$)) {
          Edge$_d$ $e$ = ADD_EDGE($n$, $n'$, $G$);
          cost($e$) = 0; }
      }
  }
  cost(Edge $best$) = $\infty$;
  while($|G(V)| > |M|$) {
    for_each_edge($e$ of $n$) {
      Graph $G'$ = COPY(G);
      $G'$ = COLLAPSE_NODES(GET_NODES($e$));
      cost($e$) = ESTIMATE_PERFORMANCE($G'$, $F$, $P$);
      if cost($e$) $<$ cost($best$) {$best$ = $e$};
    }
    $G$ = COLLAPSE_NODES(GET_NODES($best$));
  }
  return $G$;
}
```

Figure 14: Pseudo-code for the memory binding algorithm

information, and branch probabilities. First, the binding graph is created by examining the CDFG. The edges in the initial graph have a cost of zero. Solid edges of type $Edge$ are added between nodes that may share the same address space, otherwise the algorithm checks to see if it can add an edge of type $Edge_d$, which denotes nodes sharing the same memory module and not the address space. The costs are evaluated for every pair of nodes that share an edge.

As described in Section 4.1, a phase graph is extracted from the CDFG and current memory binding. Each phase is then processed to evaluate its length in cycles and to determine the number of times each loop within the phase executes. This is repeated until the expected length for the entire phase graph is evaluated. Next, the search algorithm is initiated. The search algorithm implemented is a greedy algorithm and it proceeds to identify a pair of nodes to merge by choosing the least-weight edge. Once the merge is performed, the graph is updated such that the arrays in each of the two nodes share the same memory. The function COLLAPSE_NODES is responsible for checking whether the collapsed nodes will share the address space or not, and updates the node information to reflect the choice made. Note that a node may have more than two arrays sharing the same address space and more than two arrays sharing the same memory module without sharing the address space. The algorithm continues until the arrays are all bound to the given memory modules. In addition (not shown in Figure 14 for clarity), we have implemented a simple backtracking algorithm that is used if we fail to find a solution. The algorithm is allowed to restart the search and chooses the second-best cost edge.

5 Experimental Results

We applied our memory binding algorithm to several example behaviors. The behaviors are characterized by many memory operations, control-flow operations and loops. We performed a series of experiments to demonstrate the capabilities of our algorithm. To evaluate a given memory binding, we feed the behavior, binding information, and resource information to an aggressive scheduler [17], which performs pipelining, loop unrolling, and parallel loop optimizations, to optimize the performance. The output of the scheduler is an STG and a measure of its expected number of cycles. The benchmark `InsSort` is a parallelized version of the insertion sorting algorithm. `Radix`, `Ocean` and `RayTrace` are modified kernels (such that they function with limited memory found on embedded systems) of the SPLASH-2 benchmarks [18]. `Test1` is our example from Section 3.

The first set of experiments (Table 1) compares the performance obtained by using our algorithm to that of designs where an unlimited number of memories is provided (all memories were of the same size). We provide a sufficient number of functional units such that functional unit contention is not a limiting factor. In other words, we compare our algorithm's results against the case where each array in the behavior is mapped to a different memory module, *i.e.*, parallel binding. We show that our algorithm produces schedules with performance equal to or close to the parallel case while saving in terms of memory modules. Table 1 shows that, on average, there is a 37% reduction in the number of memories at an average cost of 0.3% (no more than 1.6%) in performance.

Table 1: Performance vs. parallel case

Circuit	Parallel Memories		Our Algorithm	
	performance (# cycles)	*# mem.*	*performance* (# cycles)	*# mem*
`InsSort`	13,620	6	13,620	6
`Radix`	15,306	8	15,306	4
`Test1`	8,309	5	8,309	2
`Ocean`	129,890	16	132,011	12
`RayTrace`	236,540	64	236,540	32

The second set of experiments (Table 2) compares the binding produced by our algorithm to the best obtained from a series of random bindings. We produce a set of 15 random bindings and pick the best one. We then compare the resulting performance to that produced by our algorithm. Ideally, we would like to compare our algorithm to others in the literature. However, we did not find an algorithm that targets the same class of behaviors and, hence, have no grounds for comparison. Table 2 shows that we have an average performance improvement of 21% over all examples, when compared to the best random binding. Note that the random bindings have the same number of memories as produced by our algorithm.

Table 2: Performance vs. random case

Circuit	Our Algo. (# cycles)	Best Random (# cycles)	Worst Random (# cycles)
`InsSort`	13,620	13,620	23,071
`Radix`	15,306	18,230	33,610
`Test1`	8,309	11,643	25,672
`Ocean`	132,011	156,390	325,853
`RayTrace`	236,540	429,112	503,482

Next, we show our algorithm's ability to adapt to different resource constraints provided to the behavior (Table 3). We modify the resource constraints for benchmark `Test1` and measure the performance of each binding produced by our algorithm. We compare it to the parallel memories case. Table 3 shows that the algorithm was able to adapt to the different resource constraints without compromising performance.

6 Conclusions

In this paper, we presented a technique which performs memory binding for control-flow intensive behavioral descriptions. Experimental results demonstrate that our algorithm produces bindings that optimize circuit performance while meeting memory module and other resource constraints.

Table 3: Performance for different resource constraints for *Test1*

Resources					Our Algo.	Parallel Mem.
Add	**Sub**	**Mul**	**Cmp/Inc**	**Lsh/Rsh**	(# cycles)	(# cycles)
4	2	1	3	1	8,309	8,309
4	1	2	3	1	13,643	13,643
2	2	1	3	1	10,340	10,340
1	1	1	3	1	16,680	16,680

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Palo Alto, CA, 1990.

[2] I. M. Verbhauwehede, C. J. Scheers, and J. Rabaey, "Memory estimation in high-level synthesis," in *Proc. Design Automation Conf.*, pp. 143–148, June 1994.

[3] Y. Zhao and S. M. Malik, "Exact memory size estimation for array computations without loop unrolling," in *Proc. Design Automation Conf.*, pp. 811–816, June 1999.

[4] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, "Global communication and memory optimizing transformations for low power signal processing systems," in *Proc. Int. Wkshp. Low Power Design*, pp. 51–56, 1994.

[5] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, and J. C. Majithia, "Allocation of multiport memories in datapath synthesis," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 536–540, Apr. 1988.

[6] T. Kim and C. L. Liu, "Utilization of multiport memories in datapath synthesis," in *Proc. Design Automation Conf.*, pp. 298–302, June 1993.

[7] P. Marwedel, "The MIMOLA system: Detailed description of the system software," in *Proc. Design Automation Conf.*, pp. 59–63, June 1993.

[8] H. De Man, F. Catthoor, G. Goossens, J. V. Meerbergen, J. Rabaey, and J. Huisken, "Architecture driven synthesis techniques for mapping digital signal processing structures into silicon," *Proc. IEEE*, vol. 78, pp. 319–335, Feb. 1990.

[9] R. Cloutier and D. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," in *Proc. Int. Symp. Microarchitecture*, pp. 126–137, Dec. 1996.

[10] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, 1999.

[11] D. Thomas, *Algorithmic and Register-transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, Norwell MA, 1990.

[12] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, "An algorithm for array variable clustering," in *Proc. European Design Automation Conf.*, pp. 262–266, Mar. 1994.

[13] H. Schmit and D. E. Thomas, "Synthesis of application-specific memory designs," *IEEE Trans. VLSI Systems*, vol. 5, pp. 101–111, Mar. 1997.

[14] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, and H. De Man, "Flowgraph balancing for minimizing the required memory bandwidth," in *Proc. Int. Symp. System Level Synthesis*, pp. 127–132, Nov. 1996.

[15] F. Balasa, *Background Memory Allocation for Multi-dimensional Signal Processing*. PhD thesis, ESAT/EE Dept., K.U. Leuven, Belgium, 1995.

[16] S. Bhattacharya, S. Dey, and F. Brglez, "Performance analysis and optimization of schedules for conditional and loop-intensive specifications," in *Proc. Design Automation Conf.*, pp. 491–496, June 1994.

[17] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, "Wavesched: A novel scheduling technique for control-flow intensive designs," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 505–523, May 1999.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. Int. Symp. Computer Architecture*, pp. 24–36, June 1995.