Partitioning of Code for a Massively Parallel Machine

Michael Ball
Sun Microsystems
15 Network Drive
Menlo Park CA 94025
michael.ball@sun.com

Cristina Cifuentes
Sun Microsystems Labs
2600 Casey Ave
Mountain View CA 94303
cristina.cifuentes@sun.com

Deepankar Bairagi
Sun Microsystems
15 Network Drive
Menlo Park CA 94025
deepankar.bairagi@sun.com

Abstract

Code partitioning is the problem of dividing sections of code among a set of processors for execution in parallel taking into account the communication overhead between the processors. Code partitioning of large amounts of code onto numerous processors requires variations to the classical partitioning algorithms, in part due to the memory and time requirements to partition a large set of data, but also due to the nature of the target machine and multiple constraints imposed by its architectural features.

In this paper we present our experience in the design of enhancements to the classical multi-level k-way partitioning algorithm to deal with large graphs of over 1 million nodes, 5 constraints, and nodes of irregular size. Our algorithm was implemented to produce code for a massively parallel machine of up to 40,000 processors, and forms part of a hardware description language compiler. The algorithm and the compiler were tested on RTL designs for a next generation SPARC(R) processor. We present performance results and comparisons for partitioning multi-processor hardware designs.

1. Introduction

Partitioning is the problem of dividing a data set of an application into several parts of similar size. Many partitioning problems can be solved by representing the data as a graph and solving a graph partitioning problem. Graph partitioning divides a data set into several parts of similar size and minimizes the number of edges between those parts. Graph partitioning algorithms are widely used in industry to, for example, break a circuit into k different areas (minimizing wire length between the areas), multiply a matrix in parallel, solve sparse linear systems, place a multimedia or relational database into k clusters, and much more. It is widely known that partitioning is an NP-complete problem [15], therefore, partitioning algorithms provide approximations to a solution. In this paper we discuss code partitioning for a compiler.

Code partitioning is the problem of dividing the input data (the code representing an input program) into several parts that fit into a set of processors, for execution in parallel, taking into account the communication overhead (if any) of the target processors. A compiler that implements code partitioning represents the input data as a graph, and solves a graph partitioning algorithm. The partitioning phase can be seen as the first stage of instruction scheduling in the compiler. Code partitioning of large graphs onto numerous processors requires variations to the classical partitioning algorithms, in part due to the memory and time requirements to partition large graphs, but also due to the nature of the target architecture. Further, multiple constraints imposed by hardware characteristics of the target architecture as well as by design considerations of the compiler pose interesting challenges to adapting classical partitioning algorithms.

We have developed a partitioning algorithm for a Verilog compiler. Verilog is a hardware description language that is used to describe digital systems at several levels of abstraction. A sample Verilog program is discussed in the next section. Verilog has syntax and some structural constructs akin to traditional programming languages. A Verilog program, which is often a simulation of the hardware design described therein, makes explicit the parallelism which is inherent in digital systems. Such parallelism can be classified as code parallelism, as opposed to data parallelism, which is often found in many scientific applications.

The requirements of this compiler were to generate code for a massively parallel machine, PHASER, which can contain up to 40,000 processors. The nature of the

target machine made the partitioning problem a multiple constraint problem, where 5 different constraints needed to be addressed. The input data to this compiler are hardware RTL designs for a next generation SPARC processor, i.e., in the order of millions of lines of RTL code. A desirable requirement for the compiler was to emit code within a couple of hours. The compilation time for hardware simulators is mostly determined by the target architecture; most of the compilation time is spent on mapping the software constructs to hardware elements and meeting any hardware interconnect timing constraints.

In this paper, we present our experience in developing a partitioning algorithm for a massively parallel machine and a large set of input data. New features of our algorithm include -

- multiple invocations of a multi-level algorithm to model hierarchical machine interconnection,
- handling of multiple hard constraints[1],
- ability to partition graphs of irregular granularity[2],
- algorithmic adaptations and use of efficient data structures to provide partitions of reasonable quality efficiently, in spite of large input graphs.

Although our algorithm was implemented in a Verilog compiler targeting the PHASER machine, the design and implementation issues addressed are general enough to be applicable to other compilers and target machines.

We discuss what design decisions were made, as well as pragmatic decisions taken along the road. We first give an overview of the PHASER machine (Section 2) and explain code partitioning and the requirements of the partitioning of large graphs (Section 3). We summarize previous work in Section 4, followed by an explanation of our partitioning algorithm in the context of the PHASER machine (Section 5). We briefly describe key aspects of our implementation that make the algorithm fast and space efficient (Section 6). Section 7 describes experiments on the use of this partitioner on next generation SPARC processor designs, and its space and time performance. We conclude with a summary of our experience (Section 8).

[1] We say that a constraint is a hard constraint if overflowing it causes code generation to fail. In other words, a hard constraint cannot exceed its maximum resource allocation after instruction scheduling and register allocation.

[2] A graph is said to have regular granularity if all its nodes have similar weight or size.

2. The PHASER Machine

The PHASER machine is a hierarchical machine which consists of 10 boards connected via an interconnect (see Figure 1). The boards communicate with the host system with a Myrinet switch.

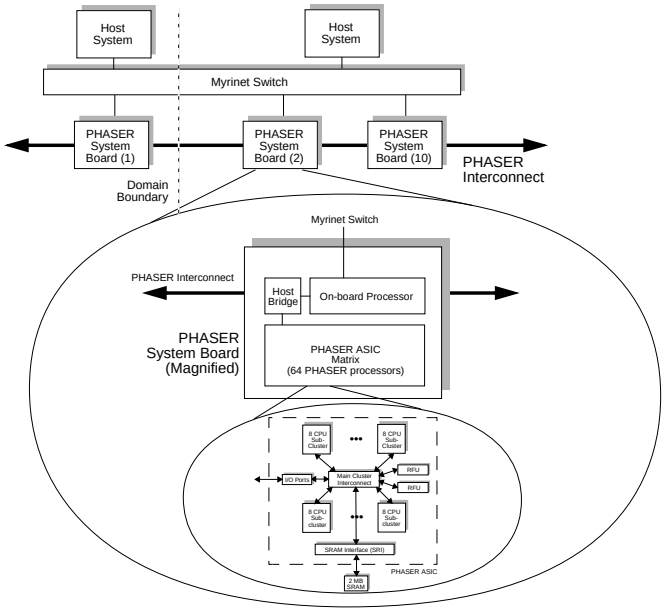

Figure 1. A High-level View of the PHASER Machine

Each PHASER board consists of 64 PHASER ASICs, which can communicate externally via a host bridge.

An ASIC consists of 8 subclusters of 8 processors each; these subclusters communicate with each other, the I/O ports, the SRAM interface, and the register file units (RFUs) via the main cluster interconnect. Each ASIC has 2 MB of SRAM.

Each processor has 1 K 8-byte words of memory, which are shared between code and data, 120 32-bit registers, 256 single bit-registers, and the CPU.

The PHASER machine hardware has three levels of interconnection. Communication overhead differs between levels. For example, board to board communication is more expensive than chip to chip communication.

3. Code Partitioning

Consider the sample 1-bit adder Verilog module in Figure 2 and its associated data dependency graph in Figure 3, which shows a graph of intermediate instructions for a compiler. In the graph, each instruction is

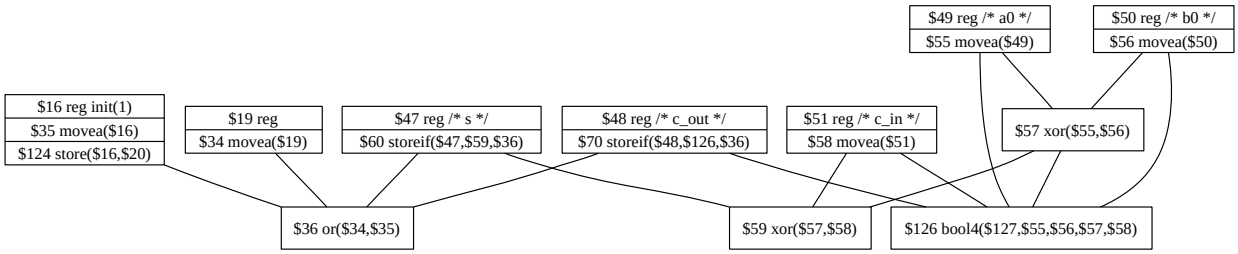

Figure 3. Annotated graph for the behavioral 1-bit adder code of Figure 1

```
module fulladder (s, c_out, a0, b0, c_in);
output s;
output c_out;
input a0;
input b0;
input c_in;
reg s, c_out;

always@ (a0 or b0 or c_in) begin
   s     = a0 ^ b0 ^ c_in;
   c_out = (a0 & b0) | c_in & (a0 ^ b0);
end
endmodule
```

Figure 2. Verilog source code for a 1-bit adder module

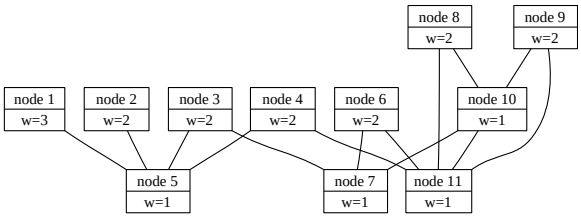

Figure 4. Graph for the behavioral 1-bit adder code of Figure 1

uniquely identified by a $N symbol, were N is an integer. For example, instruction 49 represents the register for input `a0` and instruction 59 represents an exclusive-or of the results of instructions 57 and 58. Note that the node for instruction 59 contains three edges: two that access the nodes that contain the instructions that it uses (namely, instructions 57 and 58), and one for the node that uses this instruction (namely, instruction 60). The graph contains eleven nodes that hold 19 instructions. For partitioning, we use the graph without reference to the instructions (see Figure 4; each node has been annotated with its weight (w); the weight was computed as the number of instructions in the node).

Assume we had three small processors, each containing ten slots for data and code, each slot being able to hold an instruction (weight 1). Our graph could be partitioned into two groups of nodes (partitions), of weights 10 and 9 respectively, for example:

partition 1 = {N1, N2, N3, N4, N5}
partition 2 = {N6, N7, N8, N9, N10, N11}

or

partition 1 = {N1, N2, N3, N4, N10}
partition 2 = {N5, N6, N7, N8, N9, N11}

The first group of partitions has two edges between the partitions, whereas the second one has twelve edges between the partitions. Edges between partitions represent communication between the instructions, as data used across partitions in our architecture need to be passed between the partitions by means of at least one extra instruction in each partition that is part of an edge: a `send` instruction to send the data from the partition that defines the data, and a `receive` instruction to receive the data into the partition that uses the data. Communication edges increase the number of instructions needed in each partition, which reduces the amount of memory available on a processor for other instructions to be placed in that partition (processor). Therefore the need to minimize the number of edges across partitions.

In our earlier groups of partitions, the first group of partitions is a better solution, as it takes only two edges between the partitions by introducing four new instructions. The second group of partitions would introduce twenty four new instructions. Once the `send` and `receive` instructions have been added to the code, the partitions would have a weight of 12 and 11 respectively (one `send` and one `receive` on each partition), exceeding the memory constraints of the sample processors (which have 10 slots each). Figure 5 shows the weighted graph.

Since the memory constraints for the processors are exceeded, the solution is said to be infeasible, and another solution needs to be found. In this example, three partitions (processors) are needed. A feasible solution would be as follows:

partition 1 = {N1, N2, N3, N5}
partition 2 = {N4, N6, N7}
partition 3 = {N8, N9, N10, N11}

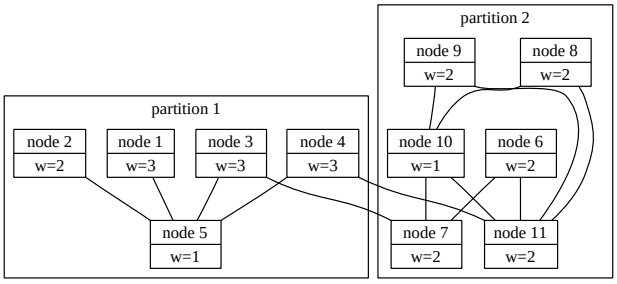

Figure 5. Partitioned graph with weights after send and receive instructions have been added to partitions (infeasible partition)

The partitioned, weighted graph is shown in Figure 6. Partitions 1 and 2 end up with a weight of 10, and partition 3 with a weight of 9.

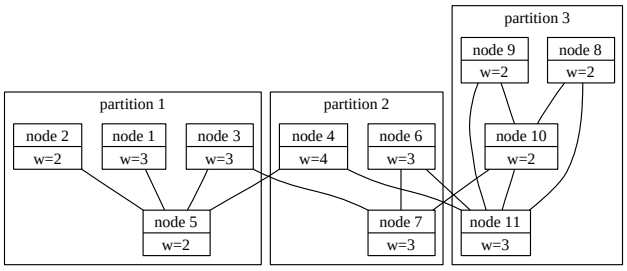

Figure 6. Final partitioned graph with weights after send and receive instructions have been added to partitions

This example illustrates the challenge of code partitioning: a complete solution needs not only to partition the input data, but also to take into account communication costs in terms of extra instructions required in the schedule, in order to find a partition that is feasible for a given set of processors.

In the previous example we were only dealing with one constraint: memory, and we were partitioning a set of data so that it fits into slots in memory. We can make the example more realistic by pointing out that a program's instructions and data are stored in different pools of memory in a processor, thereby dealing with two separate constraints: instruction memory and data memory. Such a problem is known as a multi-constraint partitioning problem.

This section's example concentrated on the graph for a 1-bit adder module, without taking into consideration the testbench associated with such a module. A complete analysis of a Verilog program requires the use of a testbench. Our 1-bit adder example with testbench produces the graph shown in Figure 7. As can be seen, the complexity of the problem increases: the graph goes up from 11 nodes to 84 nodes in total. Similar characteristics are seen on designs for complete processors.

3.1. Partitioning Large Graphs

Depending on the requirements for the compiler, a partitioning solution will have to take into account the large amounts of input data representing the graph and deal with the physical constraints of the compiler's deployment platform.

The input size of hardware designs for a next generation processor is large: 1.5 million nodes. A multiprocessor system of 2 or 4 processors involves 3 to 6 million nodes. The sheer size of these designs requires that the compiler uses efficient algorithms and data structures, since memory requirements during the compilation process can be very high.

Our experience has been that many of the classical partitioning algorithms did not scale to the size of the problem we were solving. The literature reports results for graphs of up to 250,000 nodes. The amount of time and/or memory consumed by these algorithms in the context of 1+ million nodes is prohibitive. Further, hardly any algorithm deals with multiple constraints.

We identified the following unique characteristics of partitioning large graphs:

- the graph needs to be represented in memory for the duration of the partitioning process, while avoiding access patterns that cause memory thrashing,
- the granularity of the node sizes is different, i.e., input graphs can be irregular graphs,
- "large nodes": some nodes are about the size of a partition, i.e., a large percentage of the size of a processor, and
- the compilation time needs to be "small".

Due to these requirements, we decided to use a multi-level algorithm (see Section 4) to arrive quickly at a solution. Multi-level algorithms coarsen an input graph multiple times until a small graph is obtained. The small graph is then partitioned, and the results of the partitioning are propagated back to the coarsened graphs. This allowed us to deal with the large size of the input graphs. The difference in granularity of the nodes of the input graph was dealt with in two ways: a different balancing algorithm was used during the partitioning stage, and, for large nodes, a separate extra step of the multi-level algorithm was implemented to assign such nodes to partitions ahead

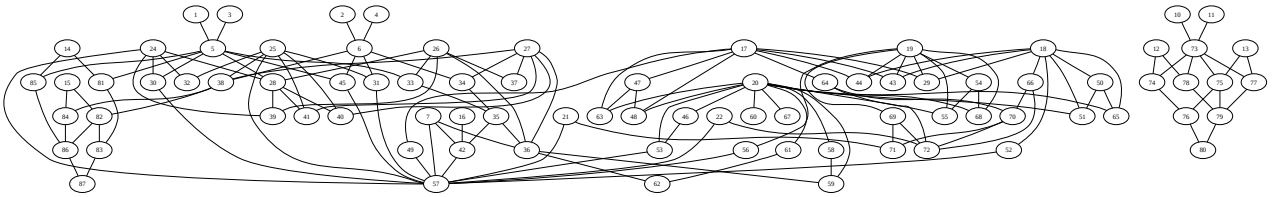

Figure 7. Graph for the behavioral 1-bit adder code with testbench

of time. We also adapted the classical multi-level algorithm to deal with multiple constraints imposed by our target platform. This algorithm is explained in Section 5.

4. Previous Work

The early partitioning algorithm that forms the basis of today's partitioning algorithms is a networking partitioning algorithm by Kernighan and Lin, commonly referred to as the KL algorithm [10]. The KL algorithm is a bisection algorithm which represents a network as a graph. Starting from a load balanced initial bisection, the algorithm moves nodes from one bisection to another if the move reduces the edge-cut[3]. This algorithm works well for a *small* set of nodes, in the order of hundreds of nodes, as it tests all nodes in the graph. An improvement to the KL algorithm is that by Fiduccia et al. [2], where by use of better data structures produces an algorithm of complexity O($|E|$).

In this section we briefly describe partitioning techniques used in general compilers and circuit partitioning. We also summarize the state-of-the-art Verilog compilers.

4.1. Code Partitioning in Compilers

Code partitioning in a compiler mostly uses one of the following types of partitioning algorithms: *critical path scheduling* or *multi-level partitioning.*

Critical path scheduling algorithms place the largest critical paths first, and the shortest critical paths last. In this way, large paths get scheduled first, followed by other paths in decreasing critical path length. Most of the critical path algorithms do not consider or do not model well the communication overhead between processors, when scheduling paths across processors. Intuitively, this algorithm is correct, however, studies have shown that it does not generate near optimal solutions for partitioning, sometimes performing worse than a sequential solution [11, 12].

[3]For a graph G with nodes N(G), an edge cut is the set of all edges having one end node in some proper node subset S and another end node in $N(G) \setminus S$.

4.2. Multi-level Partitioning

Multi-level algorithms are commonly used these days to solve partitioning problems. These algorithms are based on the observation that bisection algorithms optimize well a *small* set of nodes, therefore, the input graph needs to be "collapsed" into a smaller graph that can be efficiently and effectively partitioned.

The most widely known multi-level algorithms are those by Karypis and Kumar (the MLKP algorithm) [9], Hendrickson and Leland [6, 5] and Walshaw and Cross [20]. Each of these algorithms are based on the same premise, though different internal algorithms are used at different stages of the partitioning process. These algorithms have been implemented in tools that are available in the literature and/or reported in the literature; the tools are, METIS/hMetis, Chaco, and JOSTLE, respectively.

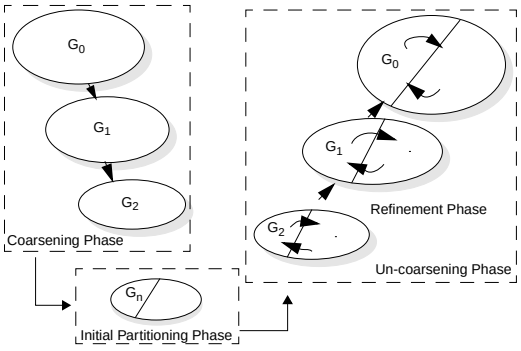

Figure 8. Multi-Level K-way Partitioning

Multi-level algorithms are best described by the process in Figure 8. An initial coarsening phase collapses nodes of an initial graph G_0 until a small number of nodes is reached in a derived graph G_n. Nodes are normally collapsed by proximity or some other function. The coarsened graph G_n is then partitioned into k partitions using an algorithm known to provide good partitions on a small number of nodes. The partitioned graph G_n is then uncoarsened, that is, collapsed nodes are uncollapsed, and the partitioned information is propagated to each of those nodes. During uncoarsening, the partition is refined, by moving nodes across

partitions if their cut set[4]can be further minimized. Once uncoarsening ends, the initial graph G_0 is fully partitioned.

Karypis and Kumar refer to their algorithm as the MLKP algorithm, a multi-level k-way partitioning algorithm. In their notation, k stands for the number of partitions needed in the solution to the problem.

4.3. Multi-constraint Partitioning

In multi-constraint partitioning, there are multiple constraints that need to be solved in order to obtain a partition. A multi-constraint variant of Karypis' MLKP algorithm is described in [8], which uses multiple priority queues to guide balancing for multiple constraints as refinement progresses.

In our work, we used a different multi-constraint algorithm to solve this problem, in part, because we had hard constraints (e.g., registers and amount of local memory used) that could not be overflowed by later code generation stages in the compiler.

4.4. Circuit Partitioning

Graph partitioning has been used widely in the VLSI community to split circuits into boards, chips, and subcircuits [16, 18]. Circuit partitioning tries to shorten the critical path and to place it in the smallest possible unit, thus minimizing communication costs. One of the important goals of circuit partitioning is to reduce the area used.

Evolutionary computing [1], and simulated annealing and tabu search heuristics [3] have also been used to aid circuit partitioning. The results of these works show better cut sets for 8 or more partitions in small to medium-sized graph (i.e., graphs of up to 150,000 nodes).

4.5. Verilog Compilers

The industry state-of-the-art tools for compiling the Verilog hardware description language are the software simulators NC Verilog by Cadence [13] and VCS by Synopsis [19]. These simulators run on traditional workstations and do not need to perform partitioning.

Hardware simulators which execute on parallel machines or FPGAs perform partitioning. The most commonly used ones are AXIS by Verisity (now integrated into SpeXsim) [17], Palladium by Cadence [14], Mentor IKOS [7], and Tharas Systems' Hammer [4]. These tools use proprietary algorithms that are not reported in the literature.

Hardware simulators are orders of magnitude faster than software simulators, however, they are not as widely used due to the expense of the hardware that is required to run the simulation on, and the need for small changes in the Verilog source code to support a design.

[4]A cut set is a set of nodes whose removal disconnects the graph.

5. Partitioning for the PHASER Machine

In this section we explain the adaptations we made to the classical MLKP [9] algorithm in order to support partitioning for the PHASER machine, which requires support for multiple hard constraints, support for large data, and support for irregular graphs.

The hierarchical nature of the PHASER machine lead us to partition repeatedly at different levels of abstraction: board, chip, and processor levels. This simplified one constraint, the communication overhead at different levels of abstraction, as by partitioning one level at a time, all nodes in that level have equal communication cost between adjacent nodes.

Figure 9 describes the ML^{3}KP algorithm, which partitions an input (Verilog) intermediate form tree representation into a PHASER system of N boards, each board of 64 chips, and each chip of 64 processors. Lines in boldface are additions or extensions to the standard MLKP algorithm. These changes are explained in the next subsections. We note that the initial partitioning requires a different balancing algorithm to that used in the standard MLKP algorithm, due to the multiple constraints to be balanced by that pass of the algorithm.

5.1. Multiple Constraints

Unlike some of the traditional partitioning algorithms, the ML^{3}KP algorithm needs to satisfy five constraints as it tries to partition an input graph onto several thousand processors. The constraints are:

- Memory (data and instruction): each PHASER processor can accommodate instructions consuming up to 8 KB memory. Similarly, a chip can hold instructions worth 64x8 KB memory and a board can hold 64x64x8 KB memory,
- Registers: each PHASER processor has 120 32-bit general purpose registers,
- Bit-register: each PHASER processor has 256 bit-registers,

```
partition_system (IF tree) {
  build graph G from IF tree
  partition by boards (G)
  rewrite IF tree based on G
}
partition_by_boards (graph G) {
  partition sram (G)
  partitions P = partition (G)
  for (all partitions p of P) do
    partition by chips (p, G)
}
partition_by_chips (partition p, graph G) {
  CG = subgraph (G, p)
  partition large nodes (CG)
  partitions CP = partition (CG)
  for (all partitions cp of CP) do
    partition by processors (cp, CP)
}
partition_by_processors (partition cp, graph CG) {
  PG = subgraph (CG, cp)
  partition large nodes (PG)
  partitions PP = partition (PG)
  for (all partitions pp of PP) do
    place partitioned nodes of PG
}
partitions partition (graph G) {
  graphs CGs = coarsening (G)
  partition P = initial partition (coarser graph(CGs))
  balance infeasible partitions (P)
  uncoarsen and refinement (P, CGs)
}
```

Figure 9. The ML^{3}KP Algorithm

- SRAM: each PHASER ASIC has 2 MB of SRAM, and

- Routing-processor instructions: each PHASER ASIC has 8192 bytes of SRAM routing processor instructions.

Some of the constraints, e.g. memory, are primarily imposed by physical characteristics of the PHASER system. The actual estimation, which is far more conservative, is discussed in the next subsection. SRAM, which is a constraint at board and chip level, also falls in the category of physical constraints.

However, constraints like registers and bit-registers are imposed by design and implementation decisions of other phases in the compiler. Registers (and bit-registers) are a constraint because the partitioner needs to be able to estimate the register requirements of a set of nodes placed in a processor. This need arises due to the hardware's use of static routing of messages. Static routing requires that instructions be scheduled to execute exactly at a clock cycle determined during instruction scheduling. Such decisions make it quite difficult to insert spill instructions, and the compiler design did not support spilling.

5.2. Building a Graph for Partitioning and Estimation of Resources

The first step of partitioning involves building a graph suitable for partitioning from the compiler's intermediate form representation: the IF tree. The IF tree is produced by the compiler frontend and intermediate optimizations are performed on it prior to partitioning. Each IF node contains an instruction or data element that represents some portion of the input Verilog program. Although Verilog supports loops, our compiler only supports loops that can be fully unrolled statically, at compile time. Each node in the graph to be partitioned consists of one or more IF tree nodes. Basically, the compiler allows a variable to reside in exactly one processor. Therefore, writes (stores) to a variable are grouped together with its definition since it is not possible to perform writes from another processor. Such groups are atomic and considered as one node in the graph to be partitioned.

The synchronization of messages and the handling of communication is carried out by the scheduling and static routing phases in our compiler, which follows the partitioning phase. These phases involve complex algorithms which are beyond the scope of this paper, nevertheless, the partitioner has to estimate resources used by these phases ahead of time.

The partitioner estimates an upper bound to the number of communication and delay instructions that will be introduced at scheduling time, knowing that the PHASER machine requires one `send` and one `receive` instruction to communicate data, and that any gaps in the generated static schedule will make use of either a `nop` instruction or a `wait` instruction (for more than one `nop`). If an instruction in a partition is waiting for data from another partition, the scheduler tries to schedule other available and ready instructions in the interleaving time slots, so that the processor for the first partition is not idle. Therefore, the partitioner conservatively estimates that each instruction will have one other instruction before it and doubles the amount of memory needed.

The estimation for registers and bit-registers is also conservative as register allocation is done after instruction scheduling. The partitioner estimates registers consumed by variables as well as temporaries. It makes a worst-case assumption about the possibility of the register allocator being able to reuse temporaries as that depends on the schedule. Further, the operands of an instruction can arrive from another processor through a `receive` and the partitioner heuristically estimates registers for such an instruction.

5.3. MLKP Algorithms and Default Values

The coarsening phase coarsens a board's IF graph (about 500,000 nodes) into smaller graphs, until we reach 500 nodes in a graph. Coarsening uses a heavy edge matching algorithm. Coarsening of graphs is done while the number of nodes in the graph is greater than 500 and while the difference between a graph and its coarsened graph is greater than 3%. In other words, we stop when significant changes in the graphs are not realized.

The initial partition is done by recursive bisection of the resulting coarsened graph. We use the greedy graph growing algorithm, as described by Karypis and Kumar [9]. This algorithm selects a node at random, places it in the selected partition, computes the edge reduction change for every unselected node on the boundary, and then selects the node with the largest change, updating the boundary as needed. If there are no more nodes connected, or the weight criterion is not yet met, it randomly chooses another node.

The uncoarsening maps back the graphs, and the refinement improves the number of edges by minimizing edges across partitions. The boundary Kernighan-Lin refinement algorithm [9] is used, which chooses random nodes from the boundary and computes the gain/loss of moving the node to each of the other partitions. The partition that results in the largest gain while still meeting tolerance and constraints is chosen. If there is none, we look at moves with 0 gain that will improve the balance. The balance is the total magnitude of the errors in the weight of the two partitions (balance is explained in Section 5.5).

5.4. Partitioning of SRAM

SRAM storage is attached to a chip, and is accessed by sending messages through the SRAM routing processor. Since there are no messages sent across SRAMs, partitioning to reduce edges is not suitable, therefore we use a different scheme. We partition data in the SRAMs with a separate pass that sorts the nodes that contain SRAM data in reverse weight order (i.e., from larger to smaller) and keep a list of SRAM bins sorted by available space. The order of bins is kept sorted using a heap, so we automatically get an even distribution of SRAM usage across the chips.

5.5. Balancing

Balancing is a key strategy which guides the partitioning process to keep the intermediate partitions within constraints as the graph is partitioned using recursive bisection. The goal of bisectioning is to divide a set of nodes into two subsets of equal size (or to a specific proportion), which are evenly balanced in all the constraints. It is very important to achieve a good balance in each level of bisectioning because the repetitive application of bisectioning to get a k-way partitioning causes any imbalance to be magnified. The tendency is to place all of the imbalance in one of the final partitions. This problem is also not trivial because the resources to be balanced for each node do not usually have amounts that are relatively proportional to each other. For example, a node may have large requirements for use of registers but little requirements for use of memory. Fortunately, there are enough nodes in a set that the algorithm should be able to find a reasonable balance in linear time complexity.

In our balancing algorithm we make use of the Euclidean distance between the resources of the two partitions. We use a normalized error function which measures the normalized vector distance of resources between the two subsets during bisection. For example, if the cumulative weight of 3 resources on one subset is (A, B, C) , the weight of resources on the other subset is (X, Y, Z), and the limiting constraints are (U, V, W), the error is:

$$E = \sqrt{(\frac{A-X}{U})^2 + (\frac{B-Y}{V})^2 + (\frac{C-Z}{W})^2}$$

To consider moving a node from one subset to another, we compute the new weights of the subsets $(A\prime, B\prime, C\prime)$ and $(X\prime, Y\prime, Z\prime)$ and a new error quantity:

$$E\prime = \sqrt{(\frac{A\prime-X\prime}{U})^2 + (\frac{B\prime-Y\prime}{V})^2 + (\frac{C\prime-Z\prime}{W})^2}$$

If $E\prime < E$ the node movement is carried out.

We found that when the cumulative weight of a resource was larger than its limiting constraint, e.g., $A > U$, we could improve the balance among the components by applying a non-linear multiplier to that component of the distance. That is, if any one error component is disproportionately large, we magnify the error still further to give the algorithm an incentive to reduce it. We used a multiplier that was non-linear; a non-linear function could be used instead.

The following example illustrates how balancing works. Our goal is to divide a set of nodes into two bisections - partition 1 and partition 2 - with constraints of (256, 128, 32) each. We have 8 nodes with the following resource requirements: Nodes 1 (32, 3, 0), 2 (16, 64, 0), 3 (32, 3, 0), 4(16, 1, 0), 5 (16, 8, 0), 6 (16, 64, 0), 7 (16, 64, 0) and 8 (16, 32, 0), where each tuple represents (memory, register, bit-register). In the following

steps, "weight (partition x)" represents the cumulative weight of all the nodes in partition x.

We begin with partition 1 empty and all nodes in partition 2. We add nodes 1, 2, 3, 4, 5, 6 to partition 1 since that improves the balance.

While adding node 6 (16, 64, 0) we have the following balancing computations: weight (partition 1) = (112, 79, 0), weight (partition 2) = (48, 160, 0)) and error $E = 0.519179$, and weight (partition 1 + node 6) = (128, 185, 0), weight(partition 2 - node 6) = (32, 96, 0) and new error $E\prime = 0.394995$. Note the non-linear penalty in registers for partition 1 as it exceeds the constraint. Still, since $E\prime < E$ the movement of node 6 is carried out.

However, when we try to add node 7 (16, 64, 0), we have old error $E = 0.394995$, weight (partition 1 + node 7) = (144, 269, 0), weight(partition 2 - node 7) = (16, 32, 0) and new error $E\prime = 0.958943$. The move of node 7 from partition 2 to partition 1 is rejected. After rejecting node 8 as well, we try to move back any node from partition 1 to partition 2 to improve balance. So node 1, 3 and 5 are moved back to partition 2, in that order. The resulting partitions so far are partition 1 = {2, 4, 6} and partition 2 = {7, 8, 1, 3, 5}.

We try again to determine if the balance of partition 1 can be improved by moving nodes from partition 2, and therefore node 3 is moved back to partition 1. The final result of the balancing phase results on the following partitions and weights: partition1 = {2, 3, 4, 6}, partition 2 = {7, 8, 1, 5}, weight(partition 1) = (80, 132, 0), and weight(partition 2) = (80, 107, 0).

5.6. Partitioning in the Presence of Large Nodes

The graph used in our partitioning process allows multiple IF tree nodes to be merged into one IF graph node. For some designs, a large number of IF tree nodes may end up being represented in the one graph node, leading to *heavy* resource requirements for that node. We commonly find a number of nodes whose resources are comparable to those available in a single processor.

```
Graph nodeID: 133447, weight = (2080, 16, 1)
 IF nodeID: $462746, opcode = MEM, width = 16384
 IF nodeID: $1252331, opcode = REFX, width = 16
 IF nodeID: $1252334, opcode = STOREIF, width = 0
 IF nodeID: $3828369, opcode = REFX, width = 16
 IF nodeID: $3828370, opcode = LOAD, width = 16
```

Figure 10. Example of a Large Node

Figure 10 is an example of a large node: a spliced array that may be placed in a processor. A spliced array is an array that has been broken up into several smaller parts of similar size, to be able to fit into a processor. The spliced array requires 16,384 bits (2,048 bytes) of storage (memory used in a processor). A reference into that spliced array, of a store and load, is grouped with the spliced array, leading to requirements of 2,080 memory bytes, 16 registers and 1 bit-register. The large memory requirement (2,080 out of the 4,096x2 bytes available in a processor) makes this node a large node.

During bisection, large nodes need to be taken into account separately because they adversely affect any bisection algorithm used to balance like nodes during the initial partitioning. One of the partitions will often end up being too heavy. Consider the following example. We are trying to partition three nodes A (40, 25, 4), B (60, 10, 6) and C (30, 25, 4) into two processors with resource constraints of (256, 32, 8) each. The combined weight of the three nodes (130, 60, 14) fits well within the constraints of a two-processor sized partition (512, 64, 16). Therefore, it would be valid for earlier stages of bisection to put them together. However, every possible combination of A, B or C is bound to exceed constraints in at least one of the processors. The balancing strategy described earlier works fine when the node weights are small compared to that of a processor. However, its efficacy breaks down when node weights are of a similar magnitude as that of a processor.

The partitioner avoids this problem by partitioning large nodes in a separate pass prior to the normal partitioning. They are distributed across the processors using a round-robin list, so that they do not create heavy partitions at the processor level. Large nodes get preallocated at both the chip and processor levels. Preallocation is performed so that bisectioning does not have to deal with these large nodes and make a chip or processor heavy beyond correction. As a general rule of thumb, a large node is one where at least one of the resource requirements is 20% of the size of the adjusted available processor resources.

5.7. Balancing Infeasible Partitions

After uncoarsening the finest graph, and prior to refinement of that graph, an extra pass over the nodes in the graph attempts to balance constraints of infeasible partitions. In this pass we only consider register and bit-register resources that are larger than the adjusted available processor resources.

We identify the heaviest node of the resource that is most imbalanced in a given infeasible partition. Then we attempt to swap this node with a node from another

partition that consumes less resources as long as the target partition remains feasible and the imbalance in the infeasible partition is reduced. Note that more than one node may need to be moved in order to make an infeasible partition feasible. This process involves exhaustive comparison of nodes. However, since only a few partitions are infeasible, typically about 3% of the total number of partitions, the time cost is not prohibitive.

6. Implementation Details

The partitioner is written in C++, as is the rest of the compiler. Its input is an intermediate code tree of (IF) nodes implemented as a linked structure on the C++ heap. Its output is the same graph allocated to processors, chips, and boards. Though such a heap structure is well suited to the variety of operations required during code generation and scheduling, it has a number of disadvantages for the partitioning process:

- the nodes contain much data not needed for partitioning, increasing the memory usage,
- linking nodes and creating sets using linked lists consumes more memory than is necessary,
- the graph is widely spread throughout memory, causing poor locality when accessing nodes and edges, and
- creating nodes and edges is a relatively expensive operation, requiring repeated heap allocation of small components.

Graphs are represented in a compact form, namely, with a set represented by a range of indices in an array. A 1:1 mapping of indices and IF tree node id's keeps track of the nodes. This is a common representation used by graph algorithms, and is much more suitable for examining the connection properties of the graph. Nodes and edges are annotated with resource consumption and, eventually, partition information. Creating such a representation is straightforward and inexpensive, but modifications to the graph, once created, are expensive. We avoid this cost by changing only the annotations once the graph has been created.

A space optimization is done during coarsening. If the change between two levels of coarsening is small (3%), the changes are combined, and the intermediate graph is discarded. This reduces storage requirements without significantly reducing refinement possibilities in the uncoarsening process.

All thresholds for partitioning (e.g., 3% coarsening, 20% large nodes, etc.) were determined empirically and can be controlled at the command-line of the compiler.

The work reported herein was the result of 2 person-years. The aim of the work was to produce reliable partitions in an industrial-strength compiler, and to provide good maintainable code. Whilst maintainability of the code is not an issue for this paper, it is important to note that such a goal was obtained in 15K lines of heavily-commented C++ code (a 50:50 ratio of comments to code).

7. Experimental Results

We report on experimental results obtained when running the PHASER compiler on a next generation SPARC processor design, and compare our partitioning algorithm to that implemented in the hMetis tool.

7.1. Experiments with 1, 2 and 4 CPU designs

Table 1 shows the results obtained by compiling several RTL designs for a next generation general purpose processor. Results for 4 designs of different sizes are shown: exu, one of the key modules inside the CPU; 1 CPU, the design for the CPU itself; 2 CPU, a multi-processor design of 2 CPUs of the same processor; and 4 CPU, a multi-processor design of 4 CPUs. For each design, the number of lines of Verilog code (LOC) are given, followed by the number of nodes in the intermediate representation (Tree nodes), the corresponding number of nodes in the graph that gets partitioned (Graph nodes), the compilation times for the whole design (Comp t) and the partitioning time (Part t), the memory usage for compilation of the whole design (Comp mem) and for partitioning of the design (Part mem). The last column, Speed, indicates the simulation speed of the RTL design expressed in terms of number of simulation cycles per second. Simulation speed is influenced to a large extent by the effectiveness of the partitioner among various other parameters.

The results were obtained in a lightly-loaded 4-processor machine with 900 MHz UltraSPARC III processors with 16 GB of main memory and 8 MB E-cache.

The compile time appears to be quadratic in the size of the design and it is dominated by the routing phase, which is indeed quadratic.

7.2. Performance comparison against hMetis

We compared the compile time performance of the PHASER partitioner with hMetis (an implementation of MLKP) [9]. Table 2 shows that even for the large designs, our partitioner is 2-4.5 times faster than hMetis. For smaller designs, the compile time difference is even

Design	LOC	Tree nodes	Graph nodes	Comp t	Part t	Comp mem	Part mem	Speed
exu	158,131	429,311	246,361	227s	31.4s	771.2MB	132 MB	114 kHz
1 CPU	1.87M	2,715,818	1,370,803	5,053s	728s	5,790MB	591 MB	38kHz
2 CPU	3.52M	5,388,950	2,713,454	19,263s	742s	11,300MB	893MB	32kHz
4 CPU	6.9M	10,067,298	4,900,235	70,312s	1,768s	25,840 MB	1,260MB	5.7kHz

Table 1. Results

	1.25 M nodes			246 K nodes	
	k=2	k=128	k=8,192	k=25	k=1,600
hMetis time	1,341 sec	3,798 sec	4,236 sec	351 sec	357 sec
PHASER time	571 sec	868 sec	1,128 sec	24 sec	54 sec
ratio	2.34	4.37	3.75	14.62	6.61

Table 2. Comparison with hMetis

higher. It should be noted that the hMetis implementation partitions the graph for one constraint while the PHASER partitioner partitions for 5 constraints, and that hMetis performed multi-phase refinement. This data was collected on a 8 processor UltraSPARC II machine with 400 MHz processors.

	hMetis	PHASER
Total time	141 sec	24 sec
Instruction count	81,730 M	12,750 M
Cycle count	186,284 M	35,548 M
D-Cache read	21,520 M	3,324 M
D-Cache read miss	1,341 M	123 M
miss/read %	6.2%	3.7%
E-Cache ref	26,516 M	3,074 M
E-Cache misses	576 M	71 M
misses/ref %	2%	2.3%
I-Cache ref	41,625 M	6,854 M
I-Cache miss	25 M	11 M

Table 3. Cache behaviour comparison with hMetis

We compared the cache behaviour of the PHASER compiler and partitioner against hMetis using cputrack on a 2 CPU UltraSparc III, 1.2 GHz Sun workstation with 3 GB of memory.

Table 3 reports data obtained using cputrack on a complete program run of hMetis and the PHASER compiler using the exu design as input data. The numbers in table are in millions.

As can be seen, hMetis executes about six times more instructions than the PHASER partitioner and takes five times as many cycles. hMetis has about 11-times more data cache read misses, 8-times more E-cache misses, and twice more I-cache misses. The first observation alone seems to indicate why the timing results for hMetis are larger. hMetis has a worse data cache miss rate, but a better E-cache miss rate.

7.3. Experiments with different processor sizes

We also experimented with how the size of the resources available in processors and chips affected the simulation speed of a given design. This experiment gives us some interesting insights into how the parallelism in the design to be simulated interacts with architectural parameters of a hardware simulation engine.

Table 4 shows that as we increase the number of processors available, and reduce the size of the processors, the partitioner spreads out the nodes to extract more parallelism. Hence the increase in simulation speed as we go from 1 to 8 processors. However, as soon as we cross the chip boundary i.e. reduce the number of processors per chip, chip-to-chip communication becomes a factor. Chip-to-chip communication is more expensive than intra-chip communication. This explains the drop in simulation speed as we move towards more chips.

8. Conclusions

In this paper we have described our experiences with the design of adaptations to the classical multi-level partitioning algorithm in order to partition code graphs of millions of nodes into thousands of processors. It was our experience that the existing literature cannot cope well with scalability; most of the reported literature deals with experimental results in the order of up to 250,000 nodes. We dealt with graphs of up to 6 million

Proc bytes	Regs	Bitregs	Procs/chip	Result
8192	120	256	64	1 processor in 1 chip, 1785.71 kHz
512	64	8	64	8 processors in 1 chip, 1937.98 kHz
512	64	8	8	8 processors in 1 chip, 1937.98 kHz
512	64	8	4	8 processors in 2 chips, 1602.56 kHz
512	64	8	2	8 processors in 4 chips, 1470.59 kHz

Table 4. Effect of processor granularity

nodes. Space and time considerations were important in our design.

Our experience revealed that balancing of multiple constraints was key to getting the partitioning to work correctly; it was also the hardest part to get right. When dealing with multiple constraints of irregular granularity, and when large nodes are present, standard bisectioning algorithms do not work well. We used an error function based on the Euclidean distance of the weights of the two bisections being considered, to cater for multiple constraints. We also pre-assigned large nodes to partitions to deal with large nodes.

The structure of our algorithm reflects the PHASER machine topology of boards, chips, and processors. This simplified dealing with one extra constraint: the different communication costs at each level of abstraction.

Last, this paper reports on experimental results for the partitioning of industrial designs onto a real machine. Many challenges were faced, the paper provides a solution to the problem.

Acknowledgements

We would like to thank Greg Wright for numerous comments to aid in the presentation of this paper, as well as Liang Chen and the anonymous reviewers.

References

[1] R. Baños, C. Gil, J. Ortega, and M. Montoya. A parallel evolutionary algorithm for circuit partitioning. In *Proceedings of the 11th Euromicro Workshop on Parallel and Distributed Processing*, Genova, Feb. 2003.

[2] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[3] C. Gil, J. Ortega, M. Montoya, and R. Baños. A mixed heuristic for circuit partitioning. *Computational Optimization and Applications Journal*, 23(2):321–340, 2002.

[4] Hammer by Tharas Systems. `http://www.tharas.com/products/index.html`. Last accessed July 16, 2004.

[5] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519–1534, 2000.

[6] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of Supercomputing*, 1995.

[7] IKOS by Mentor. `http://www.mentor.com/supportnet/ikos`. Last accessed July 16, 2004.

[8] G. Karypis. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, Univ. of Minnesota, Dept. of Computer Science, 1999.

[9] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, Univ. of Minnesota, Dept. of Computer Science, 1995.

[10] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[11] A. Khan, C. McCreary, and M. Jones. A comparison of multiprocessor scheduling heuristics. Technical Report 9307, Auburn University, Department of Computer Science and Engineering, 1993.

[12] C. McCreary, M. Cleveland, and A. Khan. The problem with critical path scheduling algorithms. Technical Report 9601, Auburn University, Department of Computer Science and Engineering, 1996.

[13] NC Verilog by Cadence. `http://www.cadence.com/products/functional_ver/nc-verilog/index.aspx`. Last accessed July 16, 2004.

[14] Palladium by Cadence. `http://www.cadence.com/products/functional_ver/palladium/index.aspx`. Last accessed July 16, 2004.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* PhD dissertation, Stanford University, 1989.

[16] N. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, Boston, Massachussets, 2 edition, 1995.

[17] SpeXsim by Verisity. `http://www.verisity.com/products/spexsim.html`. Combined Verisity's VPA and Axis technologies, after Axis merged into Verisity. Last accessed July 16, 2004.

[18] F. Vahid. Partitioning sequential programs for CAD using a three step approach. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):413–429, July 2002.

[19] VCS by Synopsys. `http://www.synopsys.com/products/simulation`. Last accessed July 16, 2004.

[20] C. Walshaw and M. Cross. Mess partitioning: a multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000.