A Hardware-Software Codesign Strategy for Loop Intensive Applications

Yuanrui Zhang and Mahmut Kandemir

Department of Computer Science and Engineering
Technical Report
CSE-09-002

January 26, 2009
A Hardware-Software Codesign Strategy for Loop Intensive Applications

Yuanrui Zhang
Penn State University
yuazhang@cse.psu.edu

Mahmut Kandemir
Penn State University
kandemir@cse.psu.edu

Abstract
Hardware-software codesign is a powerful technique that can be used to build complex embedded systems. In this paper, we propose a compiler driven hardware-software codesign strategy, aiming at facilitating algorithm architecture co-explorations. The proposed approach employs an intermediate code representation, called Loop Hierarchy Tree (LHT), to perform codesign exploration, and applies a branch-and-bound search to find a hardware-software partitioning that minimizes execution latency under the given area constraints. LHT uses fast estimation models and can be extended to handle codesign for more complex hybrid architectures. Experimental results show that our approach is successful in finding good solutions for the target applications.

1. Introduction
Hardware-software codesign is an important paradigm in designing application specific solutions and has received a lot of attention in the past (Wolf1991), (Micheli1997), (Micheli2001), (Stauffer1997). The primary task of any codesign tool/platform is to divide a given application specification (in form of either high-level code written in a language or a graph-based representation) between hardware (typically ASIC or FPGA) and software (to be mapped to the CPUs). With emergence of new FPGA architectures with soft and hard cores (Xilinx), (Altera), potential target platforms for hardware-software codesign frameworks have increased dramatically in the last five years or so. While a codesign platform can certainly be used as a standalone tool (as in the case of many existing efforts from both academia and industry (Pellerin2005), (Hark1997)), it can also be part of a larger algorithm-architecture (AA) codesign framework, which can invoke the conventional codesign tool for each algorithmic option/alternative tested (see Figure 7 as an example). Consequently, the speed of the hardware-software codesign space exploration (also known as hardware-software partitioning) can be important as well.

Many applications used in embedded computing and high-performance computing are loop and data intensive. Specifically, they are constructed using multiple (possibly nested) loops that operate on large array data. In this work, targeting such applications, we present and evaluate a new compiler-driven hardware-software codesign strategy that employs an intermediate representation called loop hierarchy tree (LHT) to facilitate the codesign space exploration. LHT is generated automatically by an optimizing compiler from a high-level description of the target application written in C/C++ and forms the base representation on which the hardware-software partitioning operates. Since our approach is compiler driven, it can also take advantage of static and during the codesign exploration process of various high-level compiler optimizations. Our strategy also employs high-level performance and area estimation modules that drive a branch-and-bound based search using LHT, with the goal of determining the minimum latency solution under specified area constraints.

We implemented our codesign framework and collected the first set of evaluation results targeting an FPGA based platform which contains multiple CPUs and hardware components. The results from six different parallel applications indicate that the proposed codesign platform generates results that are much faster than the software-only solutions (about 100 times faster in some cases) and consumes much less FPGA area (10-30% less on average) than the hardware-only solutions. This framework is very general and we also discuss how it can be extended to handle more complex hybrid platforms. Since we work on a high level application representation (written in C/C++) and employ fast yet reasonable accurate cost estimation models, we believe that our approach is particularly suitable to work with an AA codesign framework.

The rest of the paper is organized as follows, Section 2 introduces our compiler directed hardware-software codesign strategy. Section 3 describes fast high-level cost estimation models. Section 4 and Section 5 discusses extensions of our strategy and shows experiment results, respectively. Finally, Section 6 gives conclusions and points out future work.

2. Compiler directed Hardware-Software Codesign Strategy

2.1 Intermediate Representation for Codesign Space Exploration
At the core of our compiler directed hardware-software codesign strategy is an intermediate code representation, called loop hierarchy tree (LHT), which captures the inherent code structure of a function/procedure built using multiple loops. An important characteristic of LHT is that it captures the loop hierarchy of a code fragment completely and therefore enables a systematic search for codesign space exploration. Figure 1 shows the LHT in two different forms for an example funtion named Bar(). In LHT, each block of statements is placed into a leaf node in the tree, whereas each loop header statement is captured by an internal node. A special node is used as the root to represent the entire function. In this tree, only leaves contain the actual code fragments; internal nodes merely capture the loop iteration counts. Note that if-else statements can be easily handled in this representation. Specifically, as depicted in Figure 2 (a), an if-else construct can be represented by a subtree, with all conditions Ci and branches Bi being child nodes. A sub-LHT can be directly embedded into an if-else tree by substituting its Bi nodes and keeping Ci nodes along with the same statements as leaves. Figure 2 (b) gives an example of an LHT with an if-else construct.

Once an LHT is built, data flow analysis is carried out on leaf nodes (recall that only leaves contain the actual codes), restructuring the tree to reduce potential communication cost. Suppose a pair of nodes, leaf1 and leaf2, have data dependencies as shown in Figure 2 (c), and one is implemented in hardware and the other in software. Hardware-software communication cost between their subtrees can then be estimated using the loop iteration counts. If the
Figure 1. Two LHT representations for function Bar(). In (a) and (b), LHT is shown as a tree.

Figure 2. Modified loop hierarchy trees with if-else statements. In (a), an if-else statement is represented using a tree, where CI and Bt indicate condition statement and code in a branch, respectively. In (b), an if-else statement is embedded into a sub-LHT. In (c), a special node is inserted into the sub-LHT to indicate two merged subtrees that have high communication cost between them.

The estimated cost is higher than a threshold, a special node is inserted into the tree to indicate that those subtrees should be regarded as one component during the partitioning.

To further reduce communication costs in our approach, if an internal node is assigned during the partitioning process to hardware, its entire sub-LHT is also mapped to hardware. Only when an internal node is assigned to software, each of its child nodes can be mapped to either hardware or software implementation. In the following subsections, we first introduce the partitioning (codesign) for a function without any function calls, and then extend it to an entire program (which may include any number of potentially nested-function calls).

2.2 Codeign Space Exploration for a Single Function

Given area constraint, we use branch-and-bound search to find the optimal codesign solution with the least latency for a function, based on LHT representation. To facilitate the search, a data structure called mapping tree is built to systematically enumerate all possible combinational mappings/partitions for nodes in an LHT. Figure 3 illustrates a sample LHT (left) and its corresponding mapping tree (right). H and S nodes stand for the hardware and software implementation choices, respectively. The important point to emphasize is that each path from the leaf to the root indicates a potential hardware/software partitioning solution for the function (or its corresponding LHT).

Figure 3. Mapping tree and the corresponding LHT. A particular mapping has been highlighted in the mapping tree (using bold arrows). Hi and Si denote the hardware and software implementations, respectively, of node i in the LHT.

The mapping tree is generated by traversing the LHT in a breadth-first order starting from the root. If a node chooses hardware implementation, the search space for its subtree is not explored (as stated above, in this case, all sub-LHT goes to hardware). On the other hand, if a node chooses software implementation, all options for its children are systematically enumerated. Moreover, a sub-LHT rooted by a special node is considered as a whole (e.g., shaded leaf3 in Figure 3). The exploration order of children for a particular node is not unique, meaning that normally there exist more than one mapping tree for a given LHT. In the example above, if we enumerate partitions for child node2 first instead of node1, the structure of mapping tree is changed; this may affect the efficiency of the remaining part of the search. However, as each node contains only a few children, the exploration order is not expected to have a great impact in practice.

Once the mapping tree is constructed, a branch-and-bound search in a top-down breadth-first order is carried out to find the optimal codesign solution based on a heuristic pruning strategy. The search accumulates area costs when traveling down along each path. If, at an internal node, the cumulative area value is found to be larger than or equal to the provided area constraint, its subtree is not further expanded or explored (as it cannot generate any result better than what we have so far). If, at a leaf node, the cumulative area value is smaller than or equal to the constraint, a performance estimation module is invoked for the solution indicated by the path from this leaf to the root. A solution with the least latency is accepted. Note that our hardware/software codesign exploration with branch-and-bound search is invoked after potential loop transformations that can be applied by the compiler. Consequently, during exploration, the structure of loops is assumed to be fixed. In addition to this branch-and-bound based approach, we have also designed and implemented a greedy search algorithm on LHT to find near optimal solutions, which can be used where exploration speed is at most critical.

2.3 Codeign Space Exploration for the Entire Program

To extend the LHT based codesign space exploration to an entire program, function calls need to be handled. The simplest way of doing this is to process functions in a bottom-up order on the call graph representation (Cytion1991) of the code being analyzed.
so that the exploration processes for all callees are guaranteed to have been performed before that of the caller. The best co-design solution of each callee is recorded along with the corresponding area and performance costs, which are used subsequently during the callee’s co-design space exploration. Also, the bottom-up order can be obtained through a topological sort on the call graph based on the number of times functions are invoked (i.e., invocation counts). In particular, the function calls implemented outside the program as software libraries or hardware modules are assumed to have been characterized with pre-estimated costs.

Since a function may be implemented in software, in hardware or using a mix of both software and hardware, the rule that the sub-LHT should be mapped to hardware with a hardware internal node may be violated by function call statements, though it is still applicable to other code statements inside the leaf nodes. However, this fits well with the general hardware-software co-design paradigm where hardware components are allowed to call software routines as well. Also, it is worth mentioning that our technique can be used for recursive programs if the number of recursion is known beforehand (i.e., by profiling). To extend our approach to deal with arbitrary recursion is, however, in our future research agenda.

3. Cost Estimations for LHT

Compiler-driven hardware-software co-design depends crucially on fast cost estimation models, no matter what kind of search algorithm is employed. Since cost estimations take place at a very high level in the design flow, the values are usually not as accurate as those obtained by a synthesis method (Ernst1998). However, our LHT representation can help improve estimation accuracy since it separates straight line codes (in leaves) from loop iterations (in internal nodes). Specifically, for leaves, we utilize compiler based data structures and techniques for software estimations, and modeling tools for hardware estimations; for non-leaf nodes, on the other hand, we build different models for software and hardware estimations that take into account dependencies, as well as pipeline effects, if any, in hardware implementations. A unique characteristic of our high-level estimation models is that they take into account parallelism in software as well as hardware.

3.1 Leaf Node Cost Estimations

3.1.1 Software Cost Estimation

It is obvious that a leaf’s hardware area cost is the sum of callees’ area costs if there is any function call, otherwise always zero. However, it is nontrivial to obtain the estimated latency. In particular, significant static prediction complexity can be introduced not only through compiler optimizations but also because of certain architectural features like caching and pipelining. Many software performance estimation techniques have been proposed in the past, such as dynamic profiling (Ye1993), path analysis (Ernst1997), and reasoning (Shaw1989). Some of these schemes even take into account certain architectural features, as discussed in (Li1999). However, these methods usually involve a high analysis cost. Instead, we employ an operation counter based method in conjunction with path analysis for leaf nodes. By traversing the control flow graph (CFG) of the function/procedure being analyzed, all possible paths over basic blocks in a leaf node are identified. We then estimate the execution latency of every path based on the types of the operations it contains and choose the maximum among all as the cost for a leaf node. Our observation is that, since the code segment inside leaf nodes is usually short, in practice there are not too many paths to deal with.

The basic types of operations considered in our estimation model include integer addition/subtraction, integer multiplication, integer division, floating point addition/subtraction, floating point multiplication, floating point division, memory access, and function calls. Each type of operation is associated with a latency $C_t$ in terms of cycles; either obtained from the specification of the underlying CPUs or from callees’ exploration results (for function calls), as described earlier. Given the number of operations $N_t$ of each type for a particular path, the execution time (latency) $T$ can be estimated by summing up the products of the count and latency for each type, i.e., $T = \sum N_t \cdot C_t$. Since cache features have not been considered in our model yet, all memory references in the code are regarded as potential misses. We are in the process of incorporating a cache miss estimation module (based on (Fraguela2003)) into our framework.

An implicit assumption behind this software performance estimation model is that the code inside the leaf nodes runs sequentially on a CPU. In most cases, multiple units for the same function are available in a processor to support instruction level parallelism, e.g., multiple integer adders and multipliers. As a result, for a better approximation, each term in the above formula is further divided by $P_i$, which is the number of units for each type, i.e., $T = \sum N_t \cdot C_t / P_i$. In addition, data dependence analysis (Louden1997) is employed to obtain more accurate numbers.

3.1.2 Hardware Cost Estimation

Conventional hardware cost estimations carry out complete synthesis. Although the resulting estimation figures are accurate, this low-level and time consuming process cannot be employed easily by a compiler directed co-design strategy that is possibly embedded in an algorithm-architecture exploration framework. Alternatively, to obtain hardware estimations for straight line code inside the leaves, modeling tools such as (Emre2000) and (Xu1999) can be used for fast estimation, depending on hardware types (typically ASIC or FPGA). The main goal of our design space exploration is to choose a good partition at an early stage, and thus extreme accuracy may not be necessary. We have employed an FPGA-based modeling tool similar to that in (Deug2008) to estimate leaf hardware costs since our co-design strategy is evaluated using a FPGA architecture with hard cores. This tool can provide area, performance and power evaluations in a single run. Since the modeling tool needs data flow graph (DFG) as input, the DFG of code in each leaf is constructed. The compiler uses the estimated area and performance numbers output (given by the modeling tool) for the subsequent cost estimations, as shown in Figure 4.

![Figure 4](image)  
Figure 4. Interaction between the compiler and the modeling tool. Data flow restructuring is applied before feeding the leaf code into the modeling tool.

3.2 Non-leaf Node Cost Estimations

The non-leaf node cost estimations are based on the leaf node estimations. The methods for hardware and software evaluations are slightly different.
3.2.1 Software Cost Estimation

If a non-leaf node is implemented in software, its area cost is zero (the areas of its children and descendent nodes are added up automatically during the branch-and-bound search on the mapping tree when traversing down along a path). Its latency is estimated as a multiplication of its loop iteration count and its children's total latency, with consideration of two types of loop-level parallelism, the parallelism dictated by the dependencies among child loops and the parallelism dictated by the dependencies among its loop iterations. The former one is captured via a loop dependence graph (LDG) built by the compiler, as shown in Figure 5 (a). The latency is the maximum and the sum of the individual child loop latencies, for parallel and sequential executions, respectively. Using LDG, the children's latency of a non-leaf node can be expressed in terms of a combination of addition "+" and selection "max" operations. For example, in Figure 5, the total children's latency of node0 would be calculated as (node1 latency + max(node2 latency, node3 latency + node4 latency)) + node5 latency. The latter parallelism, which is restricted by dependencies across loop iterations, is also analyzed by the compiler. Figure 5 (b) depicts the dependencies with a regular distance of 4 for node0, indicating that at most 4 iterations can execute in parallel. In addition, the parallelism also depends on the number of CPUs. We define parallel degree as \( \min(\text{Parallel Iterations}, \text{Number of CPUs}) \). The latency of a non-leaf node can be expressed as (Total children latency * Loop iteration count / Parallel degree).

![Figure 5. Loop parallelism and data dependencies. (a) The loop dependence graph for the child nodes of node0. (b) The dependencies among the iterations of node0.](image)

3.2.2 Hardware Cost Estimation

If a non-leaf node is implemented in hardware, its entire sub-LHT is assigned to hardware. Hardware implementations usually apply pipeline design to loops. In this case, a non-leaf node's area cost is equal to the sum of the areas of all leaf nodes, the estimations of which are obtained from the modeling tool directly; its latency is calculated as a sum of Total Leaf Latency of all leaf nodes, where for each leaf node, \( \text{Total Leaf Latency} = \text{Leaf Latency} + \text{Leaf Total Iteration Count} \). The Leaf Latency is obtained from the modeling tool, and the Leaf Total Iteration Count is computed by multiplying the loop iteration counts of all levels of a leaf's ancestors in the sub-LHT rooted at a non-leaf node. As an example, suppose for illustration purpose, that each non-leaf node in Figure 5 has an iteration count of 10. The Total Iteration Count of leaf1 will be 1000, and the Total Iteration Count of leaves will be 100. Note that with a fully-pipelined implementation, the Total Leaf Latency is calculated as a sum of Leaf Latency and Leaf Total Iteration Count rather than a multiplication. Take node0 in Figure 3 for example. If node0 is mapped to hardware, its latency is estimated as (Total Leaf5 Latency + Total Leaf4 Latency + Total Leaf3 Latency), where Total Leaf3/4/5 Latency = Leaf3/4/5 Latency + Leaf3/4/5 Total Iteration Count. In addition, the dependencies among all leaves that belong to the sub-LHT of a non-leaf node can be analyzed to obtain more accurate estimations.

3.2.3 Cost Evaluations for the Entire LHT

Having introduced our cost estimations for leaf and non-leaf nodes, we now can take a further look at how branch-and-bound search interacts with cost estimations to evaluate a partition for an entire LHT. Before search starts, the area costs of all nodes on the mapping tree are estimated. The search then tries to calculate the total area of a partition by accumulating the area values when traveling down along its corresponding path. If the total area consumption of a partition is within the specified constraint, performance cost is then estimated in a bottom-up fashion on the LHT based on the estimation models introduced for non-leaf nodes, which is actually equal to the root's latency. Finally, the communication cost is calculated by analyzing the data flow among hardware and software loops for a partition, and added up to the total latency.

4. Extensions

4.1 Codesign for More Complex Hybrid Architectures

In contrast to conventional hardware-software partitioning schemes, our approach is easier to extend for handling hardware-software codesign, targeting hybrid architectures with more than two platforms, such as a combination of parallel CPUs, FPGAs, IBM Cell (cell) and GPUs (gpus). In this case, instead of only two options (hardware or software) for each LHT node, our approach considers multiple alternatives, resulting in a more complex mapping tree, as shown in Figure 6. Note that it is not necessary for every node in the LHT to have the same set of choices. Some may have special requirements, such as the shaded one in Figure 6, which has only two implementation choices and, thus the corresponding mapping tree for this LHT does not have uniform number of branches. However, no matter how the mapping tree looks like, our branch-and-bound search can still be carried out to find the good codesign solutions. Note also that, in this case, the cost estimation models have to be changed accordingly based on the characteristics of the underlying platforms.

![Figure 6. Mapping tree for the heterogeneous architectures with four platform choices.](image)

4.2 Hierarchical Search across Different Code Abstractions

Our hardware-software codesign approach is currently embedded within Phoenix Compiler from Microsoft (phoenix). The Phoenix Intermediate Representation (IR) has (like many other modern compilers) several abstraction levels, from machine-independent to machine dependent, including high-level IR (HIR), mid-level IR (MIR), low-level IR (LIR), and encoded IR (EIR). As compilation process proceeds, the IR goes from HIR to EIR, with more and more detailed information about the underlying architecture and runtime environment exposed progressively. The use of a low level IR
enables us to estimate the hardware/software costs more accurately. However, performing cost estimations at the low level IR can also increase the total analysis time significantly. In order to save time as well as achieve improvements over our baseline implementation, we plan to extend our current work to a hierarchical scheme across different IRs. In this approach, we would first search potential solutions over a high level IR, and pass only the promising ones to a lower level where they are evaluated again (in more detail with more accuracy) so that a final solution can be determined.

5. Experimental Evaluation

The target platform used to evaluate our compiler-directed hardware-software codesign strategy is a Xilinx Virtex-II Pro-100 FPGA with four embedded PowerPC 405 processors (xilinx). Once a codesign solution is determined, the software part is compiled targeting parallel hard cores, whereas the hardware part is translated for FPGA hardware implementations. A hardware-software interface is also inserted to enable data sharing and information flow between the hardware and software components. In addition, we incorporate an algorithm exploration loop into codesign flow, as shown in Figure 7. With application specifications, an algorithm is selected and its corresponding C/C++ implementation together with user provided constraints (including area and performance) is input to the compiler for codesign. If the latency of the solution (the partition with the least latency) found by the compiler is higher than the specified performance constraint, another algorithmic choice, if available, is selected to trigger a new codesign process.

One of our target applications is the gravitational N-body interaction problem (Malcio, 2006), which simulates the evolution of a system of N interacting bodies. There are two different matrix tiling algorithms for this application, namely plain tiling and geometric tiling (Kim, 2007). Although geometric tiling can improve accuracy, it slows down the overall execution time. Figure 8 shows the results of the codesign evaluation for both the algorithms with the gravitational kernel. It indicates that the best solution for plain tiling is 2.5 times faster than that of geometric tiling, under the same performance and area constraints (two types of area resources on FPGA, slice and multiplier, have to be considered separately during the codesign exploration along with the latency). We then map the hardware-software partition of plain tiling onto our FPGA platform to obtain the actual area consumption and performance numbers, and compare with the estimated values at high level as shown in Figure 9. The area and latency estimations have an error of 3% and $7\%$, respectively, indicating that our high level estimation models (explained earlier in the paper) are reasonably accurate in practice.

We also evaluate our codesign strategy for four data intensive parallel benchmarks FT, MG, CG, and SP, from the NAS suite (Bailey, 1994). In Figure 10, Hw stands for the pure hardware solution, Sw stands for the pure software solution, and HwSw stands for the codesign solution returned by our approach. The corresponding constraints (input to our approach) are listed in the last column. From this figure, we can see that our hardware-software solution achieves about 100 times speedup compared to the pure software implementation, while consuming less area than the pure hardware implementation. For example, in SP, it takes 8.55E+11 cycles with the software-only solution, but 7.62E+09 cycles with HwSw; and it costs 590172 slices and 16873 multipliers with the hardware-only solution, but 31780 slices and 879 multipliers with the HwSw solution. The slice and multiplier constraints given in Figure 10 are chosen based on the hardware available, and the performance constraint is currently set to be as large as possible to ensure that a solution exists.

The codesign solutions found for these NAS benchmarks indicate that such function has 10-25 loops in its LHT, with a nesting level of 4-6. By analyzing the partitions of the loops in the LHTs of these applications, we observed that the processing order of function calls has a great effect on solution quality, since resources may be consumed by insignificant functions if such functions are processed early in the process. This is addressed in our current implementation by invoking a topological sort on the call graph of the program annotated with function calling frequencies. We also observed that, with a fully pipelined hardware design, applications consisting of loops with more nesting levels can benefit more from the hardware-software codesign with limited area resources. Moreover, our codesign space exploration process is very fast, e.g., for a program consisting of 30-40 loops, it takes around 1 minute to find the best solution.

Figure 11 and Figure 12 demonstrate the success of our approach in finding solutions under different constraints, by using an image application that uses Nonuniform Discrete Fourier Transform (NUFFT) (Beogo, 2004). It can be seen from these tables that, as area resources are increased slightly, application performance can be significantly improved.

6. Conclusion and Future Work

In this paper, we have proposed a compiler-driven hardware-software codesign strategy that employs a loop hierarchy tree (LHT) representation to facilitate codesign space exploration. It applies branch-and-bound search to find a solution that satisfies the area constraint with the least execution latency. Fast cost estimation models on LHT have been built to provide evaluations for partitions. In addition, our approach can be extended to perform a hierarchical search and handle codesign on more complex hybrid architectures. Although our experimental results show the efficiency of our codesign strategy, we are planning to conduct
additional experiments with different benchmarks. In addition to computation partitioning, data partitioning will also be considered in the next version of our codegen tool.

References