# Hardware-Software Co-Design of Embedded Reconfigurable Architectures

Yanbing Li, Tim Callahan[*], Ervan Darnell[**], Randolph Harr, Uday Kurkure, Jon Stockwood

Synopsys Inc., 700 East Middlefield Rd. Mountain View, CA 94043
[*] Department of EECS, Univ. of California, Berkeley, CA 94720
[**] Silicon Spice, 415 East Middlefield Rd., Mountain View, CA 94043

## Abstract

In this paper we describe a new hardware/software partitioning approach for embedded reconfigurable architectures consisting of a general-purpose processor (CPU), a dynamically reconfigurable datapath (e.g. an FPGA), and a memory hierarchy. We have developed a framework called Nimble that automatically compiles system-level applications specified in C to executables on the target platform. A key component of this framework is a hardware/software partitioning algorithm that performs fine-grained partitioning (at loop and basic-block levels) of an application to execute on the combined CPU and datapath. The partitioning algorithm optimizes the global application execution time, including the software and hardware execution times, communication time and datapath reconfiguration time. Experimental results on real applications show that our algorithm is effective in rapidly finding close to optimal solutions.

## 1. Introduction

Reconfigurable computing using FPGAs is emerging as an alternative to conventional ASICs and general-purpose processors[1]. Reconfigurable architectures can be post-fabrication customized for a wide class of applications, including multi-media, communications, networking, graphics and cryptography, to achieve significantly higher performance over general or even special-purpose processor alternatives (such as DSPs). For convenience, we will use the term FPGA to refer to any type of reconfigurable datapath, whether implemented using FPGAs or other forms of reconfigurable logic.

Recent developments in reconfigurable architectures have demonstrated that a tightly coupled reconfigurable co-processor with a general purpose CPU can achieve significant speedup on a general class of applications[6]. An abstract model of this new class of architecture is shown in Figure 1. The architecture also contains memory hierarchy and communication channels that connect the CPU, datapath, and memory. The CPU can be used to implement control-intensive functions and system I/O, leaving the datapath to accelerate computation-intensive parts of an application. This class of architecture defines a common, reusable

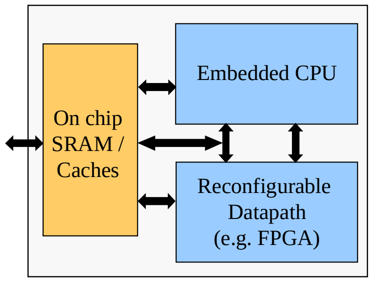

**Figure 1. The target architecture.**

platform for a wide range of applications, and potentially provides a better transistor utilization than a single CPU or combined CPU and ASIC of comparable silicon area.

To exploit the potential performance gain provided by this class of architectures, we have developed a retargetable framework named **Nimble** that automatically compiles system-level applications specified in C to executables running on these platforms. At the core of the Nimble Compiler is a **hardware/software partitioning** algorithm that partitions applications onto the CPU and the datapath. As opposed to many co-synthesis algorithms that work at moderate to coarse granularities (such as task-level and function level) and extract task-level parallelism [2][7][10], our algorithm performs fine grain partitioning at the loop and basic block levels to exploit potential **instruction-level parallelism (ILP)** to significantly accelerate important loops in the FPGA.

There have been considerable research efforts in co-design of conventional embedded hardware/software architectures containing ASICs, which we will briefly review in Section 2. However, the partitioning problem for architectures containing reconfigurable FPGAs has a different requirement: it demands a two-dimensional partitioning strategy, in both **spatial** and **temporal** domains, while the conventional partitioning involves only spatial partitioning. Here, spatial partitioning refers to physical implementation of different functionality within different areas of the hardware resource. For dynamically reconfigurable architectures, besides spatial partitioning, the partitioning algorithm needs to perform temporal partitioning, meaning that the FPGA can be reconfigured at various phases of the program execution to implement different functionality.

In this paper, we focus on the temporal partitioning aspect. The input to the algorithm is a set of candidate loops for hardware, termed **kernels**, that have been extracted from the source application. Each loop has a software version and one or more hardware versions that represent different delay and area tradeoffs. The partitioning algorithm selects which loops to implement in the FPGA, and which hardware version of each loop should be used to achieve the highest application-level performance. Key issues with this approach are:

- The partitioning algorithm must effectively capture the dynamic reconfiguration costs. This is difficult as the number of reconfigurations for one kernel depends on which other kernels may go into the hardware.
- The algorithm must integrate compiler optimizations and hardware design space exploration into the hardware/software partitioning process.
- Partitioning must be guided by various forms of profiling information to accurately assess the tradeoffs between hardware and software implementations.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, we first present an overall picture of the Nimble Compiler framework, of which our partitioning algorithm is a component, and then describe the formulation of the hardware/software partitioning problem itself. Section 4 explains the details of the partitioning algorithm. Section 5 presents the experimental results on several benchmarks.

## 2. Previous Work

Related work includes studies from general areas of hardware/software co-design and reconfigurable computing.

Earlier work in hardware-software co-design mainly focused on hardware-software partitioning. Most of the partitioning algorithms model the system based on an architectural template of a CPU (software) and an ASIC (hardware)[4] [5][7][11]. Recent work in co-synthesis has used a more generalized model consisting of heterogeneous multiprocessors with various communication topologies [2][9][10]. Although some of the above techniques use highly abstract architecture models that might be retargetable to reconfigurable architectures, none of them can represent the special characteristics of the platform such as the reconfiguration overhead or possibility of both spatial and temporal partitioning.

Some recent efforts in reconfigurable computing address automatic compilation and partitioning to reconfigurable architectures. Callahan *et al.* [1] developed a compiler for the Berkeley GARP architecture[6] which compiles source applications in C to a CPU and FPGA. They use a feasibility-driven approach that does not take performance into account during the hardware/software partitioning process. Gokhale *et al.* worked on compiling C onto reconfigurable processors but did not address the hardware-software partitioning problem directly[12].

Dick and Jha proposed the CORDS algorithm to synthesize real-time tasks onto distributed systems containing dynamically reconfigurable FPGAs [3]. CORDS uses a coarse, task-level input represented by acyclic graphs and exploits task-level parallelism. Kaul *et al.* [8] recently proposed an ILP based algorithm for temporal partitioning of reconfigurable designs that also starts with acyclic task graph specifications. The algorithm finds optimal solutions but has a very high computation cost. Its acyclic task inputs only allow a single configuration of a task and therefore use a simple configuration cost model. Both works assume a single implementation of a task in the hardware and do not explore compiler optimizations and the hardware design space to evaluate tradeoffs between different implementations of the same task.

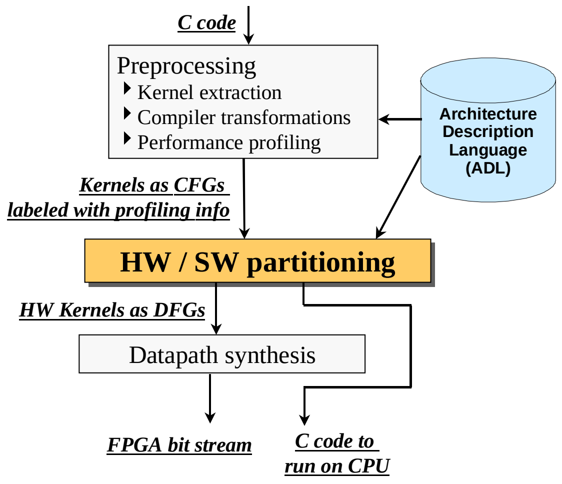

**Figure 2. The Nimble Compiler flow: an overview.**

## 3. Problem Formulation

In this section, we first give an overview of the Nimble Compiler environment. This provides a context for the problem formulation and cost function formulation of our hardware/software partitioning algorithm, which is the focus of this paper.

### 3.1 Nimble Compilation Overview

The Nimble Compiler environment depicted in Figure 2 focuses on extracting hardware kernels from the source application to accelerate on the FPGA. It is originally based on an early version of the GARP compiler[1].

The Nimble Compiler is **retargetable** and can be parameterized to the target platform described by the Architecture Description Language (**ADL**). ADL defines the components and parameters of the system such as the type of processor being used, characteristics of the reconfigurable array, memory hierarchy, etc.

Our studies show that loops represent a significant portion of application execution time, and yet are usually compact enough to implement in modest hardware resources. Therefore, the compiler focuses on finding the most profitable loops to extract out as hardware kernels. At the front end of the Nimble flow, the C program is preprocessed to extract the loop-level representations and parameters needed by the partitioning algorithm. Optimizations are applied to concentrate much of the execution time in as few loops as possible. A preprocessing step provides the partitioning algorithm a set of hardware candidates (kernels) to select from. Preprocessing not only extracts loops as kernels, but it also applies various hardware-oriented **compiler transformations** to generate multiple optimized versions of the same loop. These transformations are important because transformed code has different performance and area tradeoffs when implemented in hardware. For example, an unrolled loop requires more area in the FPGA, but it may accelerate the execution by exposing increased instruction-level parallelism. Potentially useful transformations include loop unrolling, fusion, pipelining, procedure inlining, and branch trimming, just to name a few. Profiling performed on the application and each extracted kernel to estimate the software performance, memory bandwidth need, trace behavior etc. A quick synthesis is done to estimate the delay and area of the hardware implementations.

The extracted kernels, internally represented as basic block control flow graphs (CFGs), are fed to the hardware/software partitioner which decides which kernels will go into the hardware. The selected hardware kernels are then input into our backend datapath synthesis tool to generate the corresponding FPGA bit streams, which are then used to configure the FPGA for a kernel's execution at runtime.

## 3.2 Hardware/software Partitioning Problem Formulation

We now define the hardware/software partitioning problem. The input to the partitioning step comprises two parts: the target architecture, and the set of loops/kernels extracted from the source application. The algorithm uses a fixed FPGA total size constraint as described in ADL, along with other parameters, such as configuration times and memory bandwidth.

Representing possible hardware candidates is a set of loops *L*, with each loop $L_i$ having multiple kernels $K_j$, which include an original software version and several hardware versions generated from compiler transformed code. Figure 3 shows an example with two versions for one loop. Figure 3(a) is the original CFG implemented totally in hardware. Figure 3(b) shows a transformed version after unrolling the loop once and trimming off an infrequently executed branch (marked A) by keeping it in software. *Con*, *En*, and *Ex* are overheads incurred by putting a kernel in hardware and they refer to configuration cost, hardware entry and exit costs, respectively.

The kernels and the basic blocks are labeled with profiling information obtained in the preprocessing step of the Nimble flow (Figure 2). Profiling data includes the total software execution time for each kernel, average time for each basic block and execution frequencies of basic blocks. Hardware implementation data includes the hardware area and delay for each kernel. Details of our profiling approach is beyond the scope of this paper and are not discussed further here.

As pointed out earlier, we aim to exploit ILP in loops instead of task-level parallelism. Therefore, the compiler currently only supports mutually exclusive execution of the CPU and FPGA. This simplifies the partitioner since it does not have to consider multiple loops fired off simultaneously. Loops are executed purely sequentially according to their original C specification even if pulled off onto the FPGA for acceleration.

The goal of the partitioning algorithm is to select whether to put each loop into software or hardware, and if a loop is selected as hardware, which version to use, such that the execution time for the whole application is minimized. Note that while the partitioning is generally done at loop-level, the partitioner can make basic-block level decisions by putting only a subset of the basic blocks of a kernel CFG into the hardware.

## 3.3 Global Cost Function

As the algorithm tries to maximize the overall application performance, it uses a global cost function that incorporates the hardware and software execution times, hardware kernel entry and exit delay, and hardware reconfiguration time. Equation 1 shows the global cost of all loops $T_{all\_loops}$, which is the sum of time spent in each individual loop $T(L_i)$. $T(L_i)$ denotes the total time spent in loop $L_i$, including all its iterations and entries.

$$T_{all\_loops} = \sum_{i \in L} T(L_i) \qquad (1)$$

$$T(L_i) = T_{sw}(L_i) \bullet Iter(L_i)\text{, } \textit{if Li is in software.} \qquad (2)$$

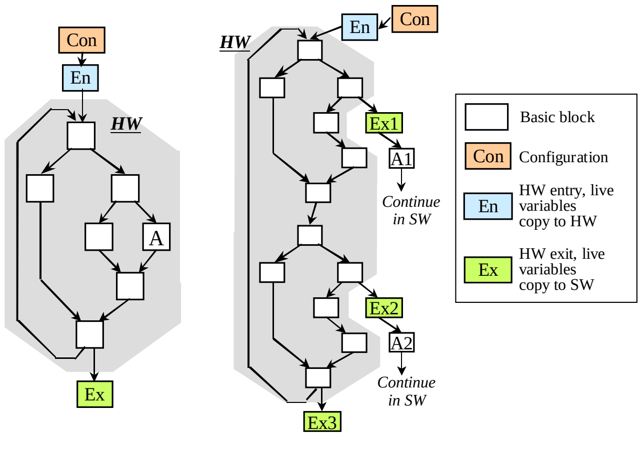

**Figure 3. Multiple hardware kernels for one loop.**

$$\begin{aligned} T(L_i, K_j) = {} & T_{hw}(L_i, K_j) \bullet Iter(L_i) \\ & + T_{sw}(L_i, K_j) \bullet Iter(L_i) \\ & + T_{sw2hw}(L_i, K_j) \bullet En(L_i, K_j) \\ & + T_{hw2sw}(L_i, K_j) \bullet Ex(L_i, K_j) \\ & + T_{config}(L_i, K_j) \\ & \quad \textit{if kernel Kj of Li is in hardware} \end{aligned} \qquad (3)$$

$$\begin{aligned} T_{config}(L_i, K_j) = {} & N_{miss}(L_i) \bullet T_{miss}(L_i, K_j) \\ & + N_{hit}(L_i) \bullet T_{hit}(L_i, K_j) \end{aligned} \qquad (4)$$

As shown in Equation 2, if $L_i$ is selected to be implemented in software only, its execution time can be characterized as the average time per iteration $T_{sw}(L_i)$ times its number of iterations $Iter(L_i)$. The computation of hardware time is more complex. Suppose we put kernel version $K_j$ of loop $L_i$ in hardware. The hardware loop time shown in Equation 3 is composed of several terms:

1. **Execution time spent in the hardware** itself. Similar to software time, it is the average hardware time per iteration $T_{hw}(Li, K_j)$ times the number of iterations $Iter(L_i)$.
2. **Execution time spent in the software** if kernel $K_j$ only implements a portion of the loop in the FPGA. (See Figure3(b) for an example of a partial loop in hardware.)
3. **Communication time** between hardware and software, which involves the copying of live variables to and from the FPGA. Since variable transfer only happens when the program enters or exits from hardware, it is obtained by multiplying the cost per transfer ($T_{hw2sw}$ or $T_{sw2hw}$) and the number of hardware entries *En(Li, Kj)* and exits *Ex(Li, Ki)*, respectively.
4. **Configuration time** of the loop on the FPGA. Unlike the previous terms which only depend on decisions made about the current loop $L_i$, configuration time depends on decisions made for other loops that interleave with $L_i$ during application execution. Some architectures (such as the GARP[6]) utilize a configuration cache to store recent configurations, so that they can be quickly reconfigured. The configuration cost for the cache miss *($T_{miss}$ (Li, Kj))* and hit *($T_{hit}$ (Li, Kj))* can be dramatically different, therefore, they must be computed separately as shown in Equation 4. The numbers of configuration cache hits and misses *($N_{hit}$ (Li) and*

$N_{miss}$ *(Li))* for a loop depend what hardware/software partitioning decisions are made for all loops.

If configuration time is not included, optimizing execution time can be reduced to locally selecting the fastest implementation of each loop that satisfies the FPGA size constraint. However, because of the complexity of computing configuration cost, the partitioning problem is NP-complete, and involves evaluating loops in a global cost function to find the optimal solution.

## 4. Algorithm Flow

Since the total number of kernels can be large for many applications, we need to deploy a heuristic algorithm to efficiently solve the hardware/software partitioning problem. The two key heuristics that we have applied are:

1. Reducing the number of loops and kernels that the algorithm needs to analyze, by focusing solely on "interesting" loops that contribute significantly to the application time.
2. For the remaining loops, partitioning them into small clusters and performing optimal selection in each loop cluster.

Based on the above heuristics, the partitioning algorithm consists of the following main steps.

1. Loop entry trace profiling (LEP). LEP generates a complete trace that records all loops entries, such that the configuration cost for all loops can be inferred.
2. Interesting loop detection (ILD). ILD screens all hardware candidate loops and only selects a subset of "interesting" loops.
3. Intra-loop kernel selection. This selects the best hardware kernel among the multiple versions of a loop implementation.
4. Inter-loop selection. Selects among loops and decides which go into hardware and software, respectively.

Steps 2 and 3 apply the first heuristic, in an attempt to cut down the number of loops and kernels to be considered. Step 4 applies the second heuristic and is the most critical step of the algorithm. The rest of this section describes these steps in detail.

### 4.1 Loop Entry Trace Profiling and Compression

When a hardware loop is entered for the first time, it needs to be configured onto the FPGA. If it is entered again before being overwritten by another loop, it does not require reconfiguration. To compute configuration cost, we need to know the exact runtime sequence of all hardware candidate loops (i.e. the entries to these loops). Loop entry trace profiling (LEP) identifies and instruments loop entries to generate a trace. The trace can potentially be huge, e.g. encoding four frames using standard MPEG-2 generates ~200M bytes of loop entry trace. LEP incorporates an online compression scheme to encode the trace. Loop trace compression not only saves storage space, but more importantly, the compact representation allows fast traversing of the trace in later steps of the algorithm. For the MPEG-2 encoding example, the trace size is reduced to several Kbytes after compression.

### 4.2 Interesting Loop Detection

While the goal of the partitioning algorithm is to select loops to implement in the FPGA to achieve maximum overall acceleration, Amdahl's law implies that we should focus on loops that represent a large portion of the application total time. We have implemented an interesting loop detector (ILD), which reports the percentage contribute a loop makes to total application time.

| ***Benchmarks*** | *# loops* | *# loops >1%* | *Total % ( >1% )* |
|---|---|---|---|
| Wavelet image compression | 25 | 13 | 99% |
| EPIC encoding | 132 | 13 | 92% |
| UNEPIC decoding | 62 | 15 | 99% |
| Media Bench ADPCM | 3 | 3 | 98% |
| MPEG-2 encoder | 165 | 14 | 85% |
| Skipjack encryption | 6 | 2 | 99% |

**Table 1. Interesing loop detection for benchmarks.**

Table 1 shows the ILD results for several benchmarks. The third column of the table shows the number of loops that contribute to more than 1% of the total program execution time. The fourth column shows the total contribution of these >1% loops. Table 1 suggests that, even though the total number of loops in an application may be large, only a small number of these loops (2—15 in the examples) contribute to most of the applications' execution time (90+%). Therefore, if we focus on these few loops, we can expect the computation cost for the algorithm to reduce significantly, yet still achieve comparable quality of results because we are accelerating most of the significant loops of the program. Furthermore, even if all loops can be accelerated by the FPGA, any speedup for insignificant loops is usually negated by the configuration overhead.

### 4.3 Intra-Loop Selection

Since each hardware candidate loop can have multiple kernels generated by compiler transformations, we apply intra-loop selection, to evaluate these multiple hardware versions, and select the best one that fits within the FPGA size constraint. This further cuts down the number of kernels to be considered in the next step—inter-loop selection. The decision of whether to put a loop in hardware or software can not be made until the inter-loop selection step. Therefore, along with the best hardware version, we also keep the original software version for further evaluation in Step 4.

The criterion for intra-loop selection is the total loop execution time, not including configuration time. This is because the number of configurations for a loop is not available until we know the hardware/software partitioning result for all loops.

Figure 4 illustrates intra-loop selection. A—D, P and Q represents several points in the hardware design space for a loop. Kernels P and Q do not satisfy the hardware size constraint. We can trim off infrequently executed branches in P and Q by keeping these branches in software to obtain the more compact implementations

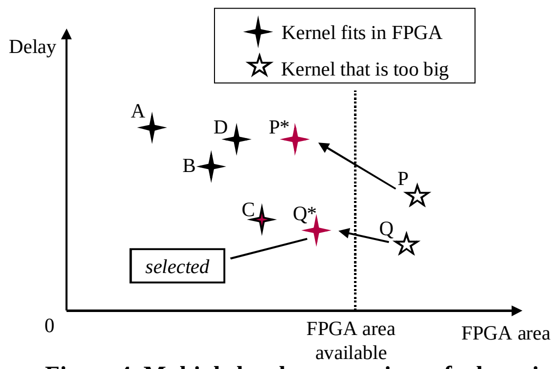

**Figure 4. Multiple hardware versions of a loop, in the area and delay design space.**

P* and Q*. For all kernels within the area limit, the fastest one (in this case, kernel Q*) is selected.

## 4.4 Inter-Loop Selection

Inter-loop selection is the most critical step of the algorithm, as the final partitioning decision has to be based on the global cost function described in Section 3.3. Selections in previous steps eliminate loops/kernels based on execution time metrics for each individual loop, while in this step, we analyze the interaction among all loops, and optimize execution time and configuration time.

Even though the number of loops (say *n*) left after steps 1 and 2 may not be very large, the number of configuration possibilities is exponential ($2^n$). We introduce a clustering technique to partition loops into small clusters to allow us to solve the partitioning problem optimally for each cluster.

### *4.4.1 Hierarchical Loop Clustering Based on the Loop-Procedure Hierarchy Graph*

Clustering of loops is based on the loop-procedure hierarchy graph (LPHG) which represents the procedure call and loop nest relations in the application. Figure 5 shows the LPHG for the wavelet image compression benchmark. A square node indicates a procedure definition, and a circular node indicates a loop. Edges into a procedure node represent calling instances to that procedure. An edge from a procedure to a loop indicates the loop is defined within the procedure. An edge from a loop *a* to another loop *b* indicates that *b* is nested inside loop *a*. There may be multiple incoming edges for a procedure, indicating multiple calling instances of the same procedure. Recursive procedures create cycles in the graph.

An LPHG captures loops and their relative positions in the application and therefore provides a navigation tool for the partitioning algorithm to traverse the loops. We define the shortest distance from a node to the root node (*main*) as the *level* of that node. We can make the following observations:

- If two loops have different first-level predecessors, they appear in a disjoint part of the LEP trace and do not compete for the FPGA configuration. For example, in Figure 5, all entries of loop FW3 appear strictly before those of RLE2. These loops can be partitioned into different clusters.
- Conversely, loops sharing common loop or procedure predecessors tend to compete with each other, and therefore should be placed in the same cluster. In the example, entries of FW3 and FW4 interleave and hence compete for the FPGA resource.

Based on the above observations, we have developed a hierarchical loop clustering algorithm based on the LPHG.We predefine a size limit for the loop clusters to ensure that the clusters are small enough for feasible optimal selection. The loop clustering algorithm traverses the loop-procedure hierarchy graph in a top-down fashion and recursively clusters loops until the sizes of all clusters are within the pre-defined limit. The algorithm works as follows.

1. Starting from the first level of the loop-procedure hierarchy, loops with a common predecessor at this level are clustered together. In Figure 5, the unshaded loop nodes are discarded after ILD. Clusters {R4}, {FW2, FW3, FW4, FW5, FW6, FW7}, {Q3, Q6}, {RLE2, RLE3}, and {E4, E3} are generated based on their level 1 predecessors.
2. If the size of any cluster exceeds the cluster size limit, we need to traverse down a level in the hierarchy and refine the clusters by grouping loops again with common predecessors at the new level. For example, the FW loop cluster has six loops. If we set the size limit at 5, we need to go down a level, to level 2, and recompute the clusters. All the other clusters are within the size limit and need no further refinement.
3. Repeat step 2 until all loop clusters satisfy the cluster size limit. In the example, at level 2, the FW loops still can not be resolved into smaller clusters and it is necessary to go down to level 3. The clustering result is shown in the figure, {FW2, FW3, FW4} and {FW5, FW6, FW7}.

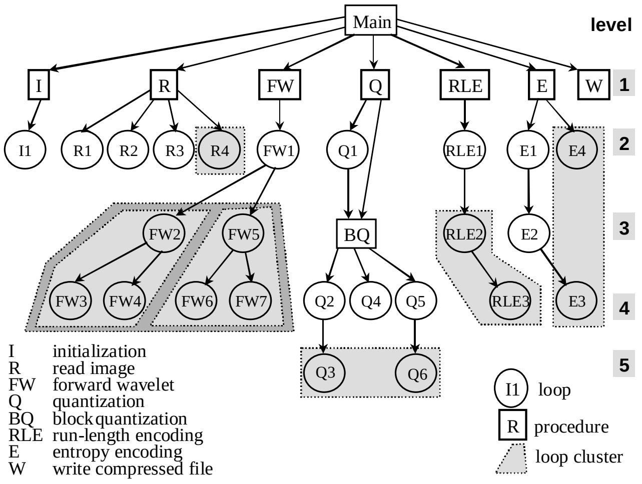

**Figure 5. Loop-procedure hierarchy graph for wavelet image compression benchmark.**

### *4.4.2 Optimal Selection in Loop Clusters*

After the loops have been partitioned into smaller clusters, our algorithm performs optimal hardware/software partitioning for each individual loop cluster. The approach adopted is to exhaustively search the solution space of all partitioning possibilities, evaluate each of these possibilities, and select the one with the best overall performance for all loops in the cluster.

To evaluate the overall performance, we must compute the number of reconfigurations needed in each partitioning possibility. This is achieved by walking through the compressed loop entry trace. The state of the configuration cache (if there is one) is taken into account to estimate the number of hits and misses.

## 5. Experimental Results

The hardware/software partitioning algorithm has been applied on real benchmarks, as part of the Nimble compilation flow. The Nimble flow takes off-the-shelf C code and compiles it onto a target architecture of a combined CPU and FPGA. The flow is fully implemented and completely automated.

In order to show the result quality and computation efficiency of our partitioning algorithm, we compare it here with a local optimization algorithm that selects loops by evaluating individual loop cost, instead of the global cost function used by our algorithm. The local optimization uses a greedy approach: if a loop shows acceleration in the FPGA, assuming it is configured once, then it is put in hardware.

We also compare the result quality of our algorithm with an absolute performance upper bound. The upper bound is obtained via the following method: For each loop (not limited to ILD loops), we use the performance of its best hardware kernel, regardless of what size it is, to estimate its hardware execution

| Benchmarks | #loops | Our algorithm | | Local-optimal algorithm | | Performance upper-bound (cycles) |
|---|---|---|---|---|---|---|
| | | CPU time (sec) | Result performance (cycles) | CPU time (sec) | Result performance (cycles) | |
| Wavelet compression | 25 | 0.17 | 1.74e+5 | 0.05 | 5.10e+5 | 1.74e+5 |
| MPEG-2 encoder | 165 | 1.92 | 7.47e+8 | 0.49 | 1.58e+9 | 7.17e+8 |
| MediaBench ADPCM | 16 | 0.08 | 7.09e+4 | 0.04 | 8.00e+4 | 7.00e+4 |
| Unepic decoding | 62 | 1.53 | 8.57e+6 | 0.28 | 1.47e+7 | 8.42e+6 |
| Skipjack encryption | 6 | 0.04 | 8.00e+4 | 0.01 | 1.10e+5 | 8.00e+4 |

**Table 2. Results of our algorithm compared to a local-optimal algorithm and the absolute performance upper bound.**

time, and we make the idealizing assumption that only one configuration is needed. The lesser of the software version time and the hardware version time (combined hardware execution time and configuration time) is used as the estimated time for that loop. This estimate provides an absolute lower bound on the execution time for that loop. This bound is optimistic: even an optimal algorithm may not always achieve this performance upper bound because of the single configuration assumption used in obtaining the bound.

The benchmarks we have used include the wavelet image compression algorithm, an MPEG2 encoder and decoder from the MPEG Simulation Group, the MediaBench ADPCM, the Unepic benchmark from MIT, and the Skipjack encryption algorithm, among other smaller test programs. All are off-the-shelf C code and compiler directly using the Nimble framework.

We experimented with the partitioning algorithm targeting two different platforms: GARP[6] and the ACEII card[13]. GARP is a single-chip architecture with a MIPS 4000 CPU, a reconfigurable array of 21 by 32 CLBs, on-chip data and instruction caches, and a 4-level configuration cache. ACEII is a board-level platform developed by TSI Telsys. It consists of a uSparc CPU and Xilinx 4085 FPGAs. There is no configuration cache on the ACEII.

Table 2 shows the experimental results of our partitioning algorithm on the GARP architecture. The table includes the performance of the partitioned design and the CPU time spent in the partitioning algorithm. These results are compared to that of the local-optimal partitioning algorithm, and the absolute performance upper bound. The results indicate that while our algorithm consumes comparable CPU time to that of a greedy local-optimal algorithm, it generates close-to-optimal hardware/software partitions in all the benchmarks shown.

## 6. Conclusions

We have presented a hardware-software partitioning algorithm that targets dynamically reconfigurable architectures consisting of a single CPU and an FPGA co-processor. The algorithm applies heuristics to achieve high computation efficiency yet finds optimal or near optimal solution in most cases. Using the algorithm in a fully automated framework on real off-the-shelf benchmarks demonstrate its effectiveness.

We plan to extend our work in the following directions: 1) instead of allowing only one loop in hardware at any time, we consider introducing multiple kernels into the same hardware configuration to improve hardware utilization; 2) improve the algorithm by more closely coupling compiler optimizations with the hardware/software partitioning, e.g. the partitioning algorithm should provide directives on what are the best optimizations to perform.

## 7. Acknowledgement

This work is partly sponsored by DARPA/AFRL under grant F33615-98-2-1317. The authors would like to thank our collaborators at Lockheed Martin ATL and UC Berkeley.

## 8. References

[1] T. J. Callahan and J. Wawrzynek, "Instruction level parallelism for reconfigurable computing," *Proc. 8th Intl. Workshop on Field-Programmable Logic and Applications*, Sept. 1998.

[2] B. Dave, G. Lakshminarayana, and N. Jha, "COSYN: hardware-software co-synthesis of embedded systems," *Proc. 34th Design Automation Conference,* 1997.

[3] R. P. Dick and N. K. Jha, "Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems," *Proc. Intl. Conference on Computer-Aided Design*, 1998.

[4] R. Ernst, J. Henkel, and T. Benner, " Hardware-software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol.10, no.4, pp.64-75, Dec. 1993.

[5] R. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, vol.10, no.3, pp.29-41, Sept. 1993.

[6] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," *Proc. FCCM '97*, 1997.

[7] A. Kalavade and E. A. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," *Proc. International Workshop on Hardware-software Co-design*, pp. 42-48, 1994.

[8] M. Kaul *et al.*, "An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications," *Proc. 36th Design Automation Conference*, 1999.

[9] Y. Li and W. Wolf, "Hardware/software co-synthesis with memory hierarchies," *IEEE Transactions on CAD*, vol. 18, no.10, pp.1405-1417, Oct. 1999.

[10] S. Prakash and A. Parker, "SOS: synthesis of application-specific heterogeneous multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol.16, pp.338-351, 1992.

[11] W. Wolf. "Hardware/software co-design of embedded systems," *Proceedings of the IEEE*, July 1994.

[12] M. B. Gokhale and A. Marks. "Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays," *Proc. FPL*, 1995.

[13] TSI Telsys, "ACE2 Card Manual", 1998.