Compiler Optimization of Embedded Applications for an Adaptive SoC Architecture *

Charles R. Hardnett
Spelman College
Computer Science
Department
Atlanta, GA 30314, USA
hardnett@spelman.edu

Krishna V. Palem
Center for Research on
Embedded Systems and
Technology
Georgia Institute of
Technology
Atlanta, GA 30308, USA
palem@ece.gatech.edu

Yogesh Chobe
Center for Research on
Embedded Systems and
Technology
Georgia Institute of
Technology
Atlanta, GA 30308, USA
ylchobe@ece.gatech.edu

ABSTRACT

Adaptive Explicitly Parallel Instruction Computing (AEPIC) is a stylized form of a reconfigurable system-on-a-chip that is designed to enable compiler control of reconfigurable resources. In this paper, and for the first time, we validate the viability of automating two key optimizations proposed in the *AEPIC* compilation framework: *configuration allocation* and *configuration scheduling*. The *AEPIC* architecture is comprised of an Explicitly Parallel Instruction Computing (*EPIC*) core coupled with an adaptive fabric and architectural features to support dynamic management of the fabric. We show that this approach to compiler-centric hardware customization, originally proposed by Palem, Talla, Devaney and Wong ([26],[27]), yields speedups with factors from 150% to over 600% for embedded applications, when compared with general purpose and digital signal processor solutions. We also provide a normalized cost analysis for our performance gains, where the normalization is based on the area of silicon required. In addition, we provide an analysis of the AEPIC architectural space, where we identify the "sweet-spot" of performance on the AEPIC architecture by examining the performance across benchmarks and computational resource configurations. Finally, we have a preliminary result for how our compiler-based approach impacts productivity metrics in the development of hardware/software partitioned custom solutions. Our implementation and validation platform is based on the well-known *TRIMARAN* optimizing compiler infrastructure [13].

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers;B.7.1[Types and Designs Styles]:Algorithms implemented in hardware,Gate arrays

General Terms: Algorithms, Performance, Design, Experimentation

9*This research was supported in part by a grant from DARPA under MCHIP PCA contract #F33615-03-C-4105

CASES'06, October 23–25, 2006, Seoul, Korea.

Keywords: Compilers, Reconfigurable Computing, System on Chip, Resource Allocation, Resource Scheduling

1. INTRODUCTION

The stringent performance requirements of embedded applications along the dimensions of power and execution time have traditionally been met via the development of application specific integrated circuits (ASICs). Nevertheless, as chip design moves into the deep-sub-micron region, fundamental changes in the economics and design cycles of chip design and manufacturing have rendered ASICs an expensive and time consuming option. *Non-recurring engineering (NRE) costs* and *time-to-market* have emerged as the twin hurdles on the path to the proliferation of ASIC-based customization in embedded systems solutions. The innovation of programming languages, compilers, and instruction set architectures (*ISA*) over the past four decades has produced great strides in increasing the productivity associated with application development in the software. Similar productivity gains must be made in the development of customized hardware if the growth and productivity benefits of embedded systems are to be realized.

Domain-specific *ISA* definitions such as those required for digital signal processing are aimed at providing many of the benefits of hardware customization, while retaining the productivity advantages of the software component of the application development process. More recently, vendors such as Tensilica [38] have developed design methodologies for the user definition of custom instructions accompanied by streamlined and automated design flows for generation of the software tool chain and ASIC implementation. This kind of design time customization requires chip fabrication for each customization cycle that in turn governs both the time-to-market as well as a significant component of the NRE costs. In addition, domain-specific ISAs must be accompanied by compilation and optimization techniques that effectively use these user-defined custom instructions.

A significant step towards achieving these goals was through the definition of the *Adaptive Explicitly Parallel Instruction Computing (EPIC)* or *AEPIC* System-on-Chip (*SoC*) architecture [26]. *Adaptive* EPIC or AEPIC is a parametric architecture introduced by Palem, Talla and Devaney [26] to help provide a reconfigurable or adaptive SoC architecture for custom embedded computing, while providing extremely fast embedded application development and compilation times: on the order of minutes and comparable to application development times in the context of a standard (non-adaptive) ISA. Thus, the goal was to provide support for customization

while dramatically lowering the *non-recurring engineering costs* and *time-to-market* associated with COTS reconfigurable SoC architectures providing a judicious amount of architectural and micro-architectural support. Specifically, AEPIC generalized the concept of a traditional ISA that mediates between a programming model and its compiler, and the underlying micro-architecture circuitry, by providing an abstract interface to the reconfigurable component by treating it as an "elastic" extension to the ISA. Subsequently, Palem, Talla and Wong [27] defined a compilation "flow" and specific optimizations for compiling to the reconfigurable components. The value of all of these concepts were established by Talla in his dissertation [36] wherein the compiler optimizations were applied to canonical embedded applications by hand.

In this paper, we present our approach to realizing this goal based on the work of Palem, Talla and Devaney [26]. The contributions of this paper are:

1. We revise, implement, and validate the algorithms and optimizations for configuration allocation and scheduling of custom instructions on our AEPIC reconfigurable SoC. This is based on work by Palem, Talla and Wong [27].

2. We evaluate the value of applying these algorithms to a range of embedded system benchmarks and demonstrate significant performance gains Specifically, the automation in this paper yields speedups:

   (a) Ranging from 50% to as high as 600% over a conventional EPIC processor with 6 functional units, realized using the HPL-PD framework [19].

   (b) Ranging from 25% to over 500% in the context of a TI TMS320C6713-300 DSP processor.

3. We also show that when these gains are normalized by the area of the respective architectural implementations, the gains in running time or speedups are evident:

   (a) Ranging from 10% to as high as 500% over a conventional EPIC processor with 6 functional units, realized using the HPL-PD framework [19].

   (b) Ranging from 5% to over 250% in the context of a TI TMS320C6713-300 DSP processor.

4. By exploring the architecture space characterized by the areas used by the EPIC component and the AEPIC co-processor of AEPIC, we determine that the gains stated above can be sustained and grow with increasing area investments till a point of diminishing returns is reached. Specifically, we show that this occurs when the EPIC component in the AEPIC processor or SoC is 66% of that used to realize a stand-alone EPIC processor, whereas the adaptive component is no greater than 75% of the Xilinx Virtex II FPGA.

1.1 Roadmap for the rest of the paper

In the next section (Section 2) we present a brief survey of related work in terms of configurable architectures and the tools for utilizing these architectures. In Section 3 we describe our compilation flow and the relevant algorithms for managing and effectively leveraging the AEPIC architecture. The three major steps (i) Partitioning (Section 3.1) (*ii*) Configuration selection (Section 3.2) and (*iii*) Configuration allocation and scheduling (Section 3.3) are described as well. In Section 4 where we present the speedup obtained from AEPIC architectures through our compile-time optimization techniques, we also study area-performance trade offs in AEPIC architectures. Finally in Section 5, we present avenues for future research.

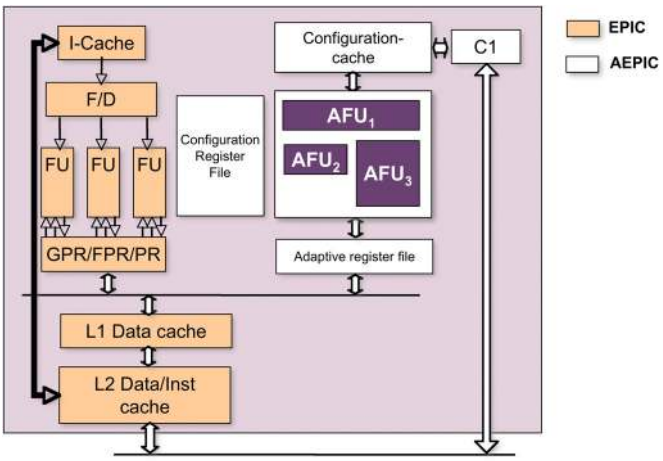

Figure 1: The *AEPIC* architecture is shown. We have outlined the components of the AEPIC architecture that are part of an EPIC architecture as specified by HPL-PD, and the components contributed by AEPIC.

2. RELATED WORK

In this section we describe the work that has influenced the theme of our work in the way we formulate solutions to the problems we will present. We compose a context for our work by presenting the origins of AEPIC, the basis of our allocation and scheduling algorithms, and similar tools for adaptive architectures.

2.1 Origins of AEPIC

As NRE costs and time-to-market considerations have become dominant in SoC designs, commercial platforms have emerged into two paths to support custom hardware coupled with the host microprocessor. The first was a reconfigurable fabric (see for example [23][29][12][34]), and the second was masking programmable logic blocks in platforms now referred to as structured ASICs (see for example [22][11]).

AEPIC is an architectural model that extends the *Explicitly Parallel Instruction Computing Architectures* (*EPIC*) [30] design philosophy, wherein instruction level parallelism (*ILP*) is explicitly communicated by the compiler to the hardware. AEPIC [26] [36] extends this model into reconfigurable SoCs, and is shown in Figure 1. Here AEPIC has support for a reconfigurable fabric, which can be dynamically configured as a collection of Adaptable Functional Units (AFUs). The application software is viewed as having program regions targeted for customization, and these regions are then executed on AFUs. The AFUs are configured by a stream of bits called a *configuration*. AEPIC specific instructions are executed on the EPIC processor to send the configuration stream to the adaptive fabric. Similar instructions are executed to manage configurations within the *configuration memory hierarchy* composed of configuration registers, configuration cache, and the C1 cache. These instructions provide the ability to load and unload configuration data to and from specified levels of the memory hierarchy. The configuration registers are used as handles for configurations

within the configuration memory hierarchy. These handles are typically parameters to control instructions to begin execution, set up input/output between the EPIC processor and adaptive fabric, and to query status information about the custom instruction execution. The cache hierarchy helps to minimize the long latency that results from the loading/unloading configurations. In the next subsection, we present our basis for the compiler algorithms to support customization.

2.2 Architecture Allocation and Scheduling

Allocation and Scheduling have been at the forefront of compiler technology over the past several decades. Due to the prevalence of register allocation and instruction scheduling strategies and the massive engineering tasks for developing compilers, our algorithms for customization make use of technology currently found in modern optimizing compilers [9][15]. Our allocation strategy is based on concepts of register allocation and graph coloring [8] [7]. We treat the reconfigurable fabric as a register file via the configuration registers, where configurations are allocated to K Adaptive Elements (AEs) to support the execution of the custom instruction on an AFU, where an AFU is a collection of cooperating AEs. The allocation of K AE resources is a constrast to the 1 data register resource required to store a data element. This makes the allocation problem a *graph multi-coloring* problem which is a more general problem than a *graph coloring problem*. We provide details for this algorithm in Section 3. Our custom instruction scheduling strategy is based on the scheduling problem and conjectures presented by Talla, et al. [28]. The scheduler is based on greedy-list scheduling [14] with ranks [25].

2.3 Tools for Customization and Adaptive Architectures

We use custom instructions to extend the base *ISA* of an embedded microprocessor under a variety of physical and performance constraints. Companies such as Tensilica [38] and ARC [2] pioneered commercial models of an approach for creating custom implementations of microprocessors with an extended *ISA*. In addition, vendors started producing platforms with microprocessors coupled with reconfigurable fabrics to host custom instructions/coprocessors. All of these efforts rely on sophisticated program analysis or user definition to identify custom instructions. See [40][10] for a good overview of current and past research in custom instruction discovery. Further, Clark et.al [10][21] describe a compiler infrastructure for automating *both* the discovery of custom instructions and its use within an optimizing compilation framework from standard high-level languages. The *AEPIC* compiler framework described here complements this body of work by automating the optimization phases of the compiler devoted to the allocation and scheduling of these custom instructions, and to the authors' knowledge this has not been investigated where the allocation and scheduling are automated for an AEPIC class processor with dyanamic reconfiguration ability, and region-sized custom instructions. Where a region is considered a loop-body, function-body, etc.

Other related tools include *PICOExpress* and *PICO Flex* from the Program-In-Chip-Out (PICO) project at HP Laboratories who pioneered an approach that leverages well known program transformations such as software pipelining to synthesize non-programmable hardware accelerators and a custom companion Very Long Instruction Word (VLIW) processor to implement loop nests found in ANSI C programs [31]. More recent work on C-based design flows have evolved under the umbrella of *algorithm-based synthesis* recognizing the differences from hardware based behavioral synthesis.

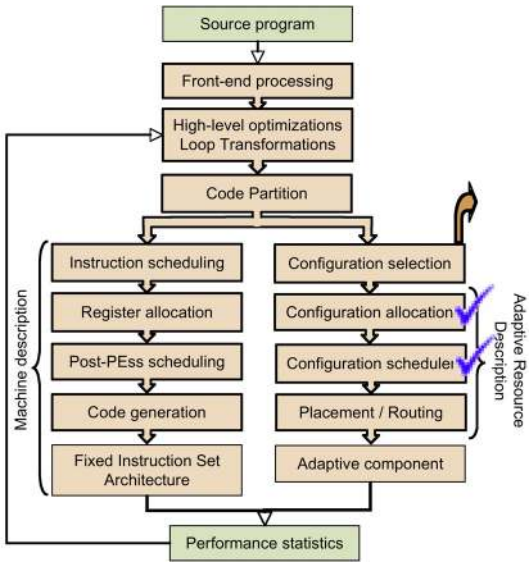

Figure 2: The *AEPIC* compiler flow where the boxes marked with a $\surd$ mark are the topic of this paper and are automated. The flow on the left-hand side of the figure is a traditional EPIC compiler-flow and is automated. The code partitioning and other modules of the right-hand side are performed by-hand.

Significant examples include DEFACTO [33] [32], SilverC [37], SpecC [35], ImpulseC [1], and HandelC [6]). Compilation of applications or kernels described in these languages are focused on algorithm based synthesis to HDL implementations that can subsequently be processed by standard EDA tool chains typically targeting FPGA devices. These technologies also complement our *AEPIC* compilation flow by providing algorithms for the generation of custom instruction implementations that are allocated and scheduled by subsequent phases of the *AEPIC* compiler. These compilers enable compilation to an FPGA, but are not designed to compile to *AEPIC* and take advantage of the architectural support *AEPIC* gives to the compiler. In addition, these tools rely on synthesis tools during the compilation flow, which ultimately reduces the productivity of these approaches.

3. COMPILER OPTIMIZATIONS FOR CUSTOMIZATION

Our compiler flow is shown in Figure 2. The input program or application is processed by a standard front-end, followed by a partitioning phase. The *partitioning* module examines the Program Dependence Graph (PDG) of the application for frequently executed data parallel regions. We felt that data-parallel regions of the application provided the best opportunity for performance gains and make better use of the reconfigurable resources. These regions are used to create a set of candidate custom instruction regions. Conceptually, the compilation flow splits the program into two portions: the left side (Figure 2) to handle traditional threaded code and the right side to handle custom instruction regions. Our work reported in this paper is on the right side, where it starts with the *configuration selection* module. The configuration selection module (Figure 2) selects the "best" machine configuration for a given custom instruction candidate region that we perform by hand. The choice is based on several criteria including latency, area, and

power. The PDG is then updated to include custom instruction opcodes to replace the custom instruction regions. The main contributions of this paper start at this point, where the *configuration allocation* step examines the PDG (annotated with custom instructions) and generates a allocation/de-allocation mapping of custom configurations to the adaptive resources. The *custom instruction scheduling* step is used to determine a feasible schedule of portions of the PDG that correspond to a the given Data Dependence Graph (DDG). The schedule associates DDG nodes with AFUs at a given time t.

3.1 Partitioning

Partitioning; done by hand, is illustrated in Figure 3, for the FFT benchmark. In Figure 3, these custom instruction candidates are parts of the main loops in the implementation to perform the FFT. Our hand-partitioning requires application profiling to generate an annotated dynamic call-graph to indicate the "hot-spots" of the application. The "hot-spots" are fine-grained regions such as loop bodies or coarse-grained regions such as entire functions. The "hot-spots" are then analyzed for the following favorable characteristics:

1. The region must be an entire loop region or body of a loop region,
2. The region must have data parallelism (no loop-carried dependencies), and
3. The region must not contain system calls or application subroutine calls.

The partitioning phase involves applying well-known loop and data transformations to remove loop-carried dependencies [3]. For this paper, we have applied the transformations by hand when needed. Transformations such as loop fusion, (look at textbook) carried dependencies enables us to extract These loop transformations have been used on parallel computing systems to enable more loop-level parallelism and increase data locality. In addition, subroutine inlining is used to remove subroutine calls from the bodies of the region[39].

3.2 Configuration Selection

Configuration selection constructs machine configurations from carefully selected primitives. The machine configurations are the $AFUs$ that support the execution of custom instructions identified in partitioning. The configurations are built from primitive operations such as adds, multiplies, and shifts. Each operation is characterized by its area, power, frequency, and latency. These elements are composed in different ways to produce configurations with the same functionality but different area, power, frequency, and latency features. All of these configurations are organized in a configuration library(see Figures 2 and 4). Therefore, each custom instruction is supported by multiple configurations and the selection of a configuration is the first step in meeting the performance requirements of the application.

3.3 Configuration Allocation

This subsection discusses Configuration Allocation, where the PDG annotated with custom instruction regions is given as input. The problem is to find a allocation strategy that enables custom instructions to effectively share the adaptable resources during the execution of the program. We have formulated the solution to this problem within the context of register allocation. This allows us to reuse the traditional compiler modules for supporting register allocation including live-range analysis, pruning, and live-range splitting.

Table 1 summarizes the relationship between the allocation problem we solve for the adaptive fabric and the traditional register allocation problem. In our scenario, we generate a hypergraph that represents the allocation of custom instructions to resources of the adaptive array. Our allocation algorithm views the adaptive computational resources as a single-dimensional array. For this reason, our allocation algorithm determines whether resources are available to be allocated instead of concerning itself with the layout and shape of the allocation. In doing this, the allocation algorithm assumes the fabric can be partially reconfigured during the execution of the application. Therefore, custom instructions with overlapping live-ranges should be allocated to non-intersecting sets of AEs. Custom instructions with non-overlapping live-ranges may share resources. We form an interference graph to identify these live-range characteristics just as with register allocation algorithms.

A formal description of the configuration allocation problem as a graph multi-coloring problem is:

Let $G(V,E,\omega)$ be an undirected graph where ω is a weight function, $\omega : v \rightarrow Z$ where $v \in V$.

Let $C : v \rightarrow S$ be a function on the vertex set such that $S \subset 1,\ldots,K$.

Therefore, C is a valid multi-coloring of the graph if $|\, C(v) \,| = \omega(v)$ and $\forall e \in E$, where $e = (u,v)$ and $C(u) \cap C(v) = \emptyset$.

Algorithm 1 Graph Multi-Coloring Configuration Allocator (GMCA)

```
Input: PDG          :Program Dependence Graph
Input: TR           :Total allocatable resources
Input: HEU          :Heuristic for ordering allocations
Output: IG          :Interference Graph(G)
Output: AHGRAPH     :Allocation Hypergaph
  IG ← BuildInterference(PDG) // Live-Range Interferences
  ComputePriorities(IG, HEU) // Heuristic ordering
  while IG not empty do
    SaveUnconstrainedOps(PDG, IG)
    v ← HighestPriorityCustomInstruction(IG)
    if IsColorable(v) then
      Color(v, IG, AHGRAPH, PDG)
    else
      if SelectAlternative(cop) then
        continue
      else
        splitCost ← SplitCost(ig, v)
        spillCost ← SpillCost(ig, v)
        if splitCost ≤ spillCost then
          Split(ig,v)
        else
          Unallocate(v)
          RemoveCop(v)
        end if
      end if
    end if
  end while
  ColorUnconstrained(PDG, IG, AHGRAPH)
```

Our configuration allocation algorithm is presented as Algorithm 1. The algorithm builds live-range interferences [7] between pairs of custom instructions to determine where the overlapping live-ranges exist, this is done in the *BuildInterferences* function which uses the def-use chains of custom instructions to compute live-ranges

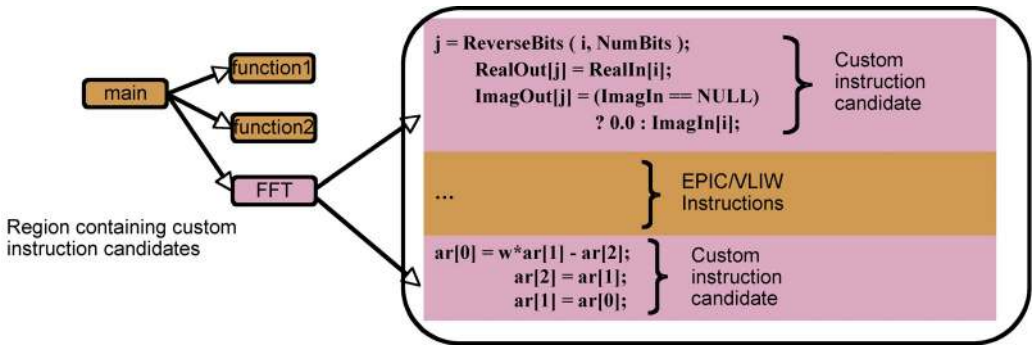

Figure 3: A demonstration of the effects of partitioning shows the call-graph with several regions, where the FFT region is the focus for custom instructions. The right-hand side shows candidates for custom instructions that are parts of the bodies of loops in the FFT region. Due to space constraints, we have not shown the entire custom instruction(s).

Register Allocation	Configuration Allocation
Each virtual register requires *one* physical register	Each virtual configuration "register" requires K adaptive resources
Allocation processed in units of *one* register	Allocation processed in units of K adaptive resources
Graph coloring *one* color per physical register	Graph coloring K colors per virtual configuration "register"
Represented by a *digraph*	Represented by a *hypergraph*
Live-range is a path of nodes that definitions and uses of a variable	Live-range is a path of nodes that begins with the load of an instance of a configuration and ends at the uses of that instance
Register pressure is where the maximum number of registers is required by the program region	Resource pressure is where the maximum number of adaptive resources is required by a program region
Register pressure may be reduced by splitting/altering conflicting live ranges producing spill code	Resource pressure may be reduced by splitting/altering conflicting live ranges, selecting alternative configurations, or executing on the core processor (unallocating)

Table 1: Register Allocation vs Configuration Allocation. This table summarizes our motivation to treat the configuration allocation problem as a generalized register allocation problem to reuse the compiler's register allocation infrastructure.

and determine conflicts. Allocation of colors is controlled by priority queue controlled by the *ComputePriorities* function. Our heuristic is designed to favor efficient configurations over less-efficient configurations. This is done using a normalized $latency/area$. This factor allows us to judge the amount of parallelism achieved. Low latency and high area indicate high-levels of parallelism. These are at the head of the priority queue followed by balanced configurations with ratios of 1. The custom instructions (represented by v) are then processed in order using the modified graph multi-coloring algorithm presented in earlier work [36] [7]. Each custom instruction is assigned a set of colors (representing AEs) when there are no conflicting neighbors in the IG. When conflicts arise, this algorithm incorporates the following ideas to resolve them:

Live-Range Splitting: The live-range for all but one of the conflicting operations is split, and then compensation code is generated to handle the unloading and loading of the configuration who's live-range is being split. The cost of live-range splitting is directly proportional to the amount and additional latency incurred by the compensation code. This compensation code will ensure that conflicting configurations are in the configuration cache, and have them loaded with other configurations. This may affect the schedule and timing during scheduling.

Alternative Selection: The library of configurations is consulted for a replacement configuration that uses less area and still achieves the latency and power requirements.

Unallocate: The custom instruction is not allocated to the adaptive array, and is executed on the core processor. This is similar to spilling registers to memory because it uses another resource (the fixed processor) and it is likely to decrease performance.

In Figure 5, we show how a hypergraph is used to represent the configuration allocations. In this figure, the AE's are used to create $AFUs$ to support the custom instructions invoked by *exec* operations and corresponding virtual configuration registers, *vcr*'s. The resource mapping shows the set of AE's that form the AFU for each custom instruction. This formulation allows our algorithm to determine which resources are shared between $AFUs$, which will drive custom instruction scheduling decisions in the next phase.

3.4 Custom Instruction Scheduling

The custom instruction scheduler leverages the instruction scheduling infrastructure of a traditional compiler. As shown in Table 2, the framework of the problem and the solution are similar. This enables us to reuse the EPIC compiler scheduling infrastructure. The added variability of AFUs in custom instruction scheduling is what elicits the difference, which is manifested in the way resource

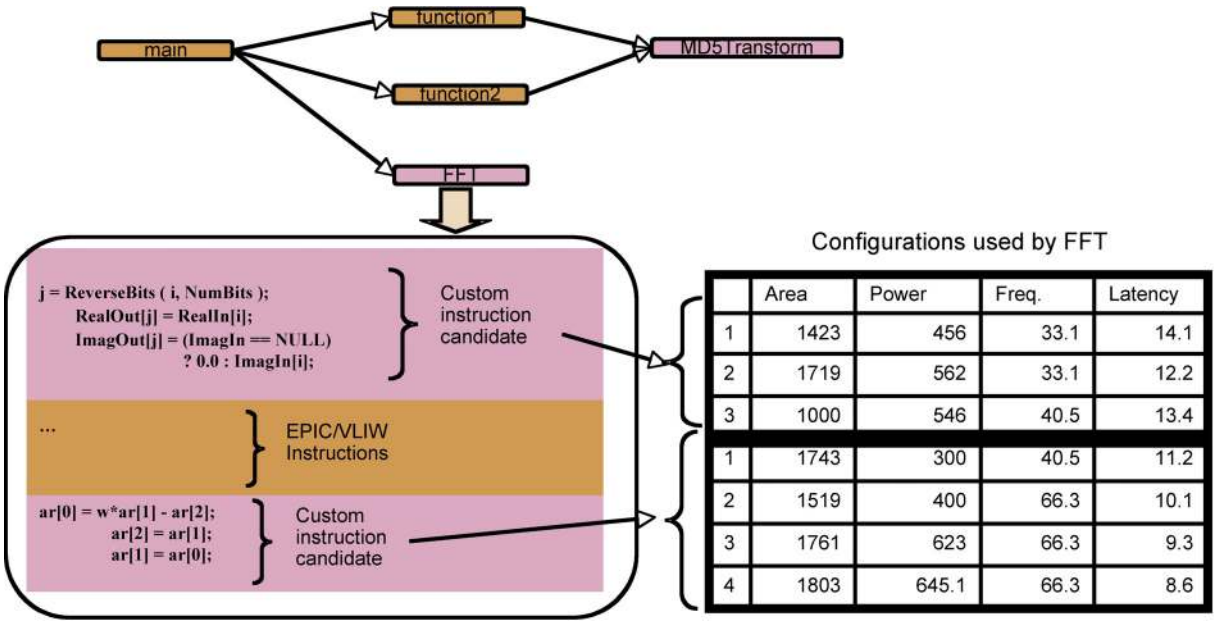

Figure 4: This demonstration of Configuration Selection shows where the custom instructions may be mapped to several configuration options with trade-offs. These trade-offs are area, power, frequency, and latency

Traditional Instruction Scheduling	**Custom Instruction Scheduling**
Fixed number of functional units over time	*Variable* number of functional units over time
Instructions are given start times	Custom instructions are given start times
At any time T, instructions are bound to functional units and there is no more than one instruction bound to a given functional unit	At any time T, instructions are bound to AFUs and there is no more than one custom instruction bound to a given AFU
Scheduled Instructions obey precedence and latency constraints	Scheduled custom instructions obey precedence and latency constraints

Table 2: Traditional Instruction Scheduling vs Custom Instruction Scheduling. This table shows that our scheduling problem is not unlike a traditional instruction scheduling problem for a VLIW processor.

availability is done. This means that the framework for instruction scheduling is identical with one relatively minor change.

Formally, the scheduler accepts as input a Program Dependence Graph G_{pdg}, with a subset of nodes I_c that correspond to the custom instructions. As result, the proposed problem is to find a legal schedule such that:

Latency: Given instructions i, j if $i \prec j, \sigma(i) + \pi(i) < \sigma(j)$, therefore, instruction i appears in the schedule before instruction j, and instruction i completes execution before the start of instruction j.

Resource Assignment: For custom instructions i, j if $(\psi(j) \cap \psi(i)) \neq \emptyset$ then $[\sigma(i), \sigma(i) + \pi(i) - 1] \cap [\sigma(j), \sigma(j) + \pi_i - 1] = \emptyset$. If the instructions i and j share resources, then the instructions must not have overlapping execution intervals.

where $\forall i \in I_c$

1. $\sigma : i \rightarrow N$, where N is the set of start times,
2. $\psi : i \rightarrow S$, such that $S \subset 1, ..., T$ where there are T slices of the reconfigurable fabric.
3. $\pi : i \rightarrow \mathbf{Z}$, defines the latencies of an instruction

Algorithm 2 Custom Instruction Scheduler

Input: DDG : Data Dependence Graph
Input: AHGRAPH : Allocation Hypergraph
Output: SCHED : List of tuples (start time, custom instruction) sorted by start time

Place ops on ReadyList
while ReadyList is non-empty **do**
  $cop \leftarrow$ ReturnHighestPriorityCop(DDG)
  $stime \leftarrow$ EarliestTimeforScheduling(cop)
  while not ResourcesAvailable($stime$, cop, $SCHED$, $AHGRAPH$) **do**
    $stime \leftarrow stime + 1$
  end while
  $schedule$(cop,stime,$SCHED$)
end while

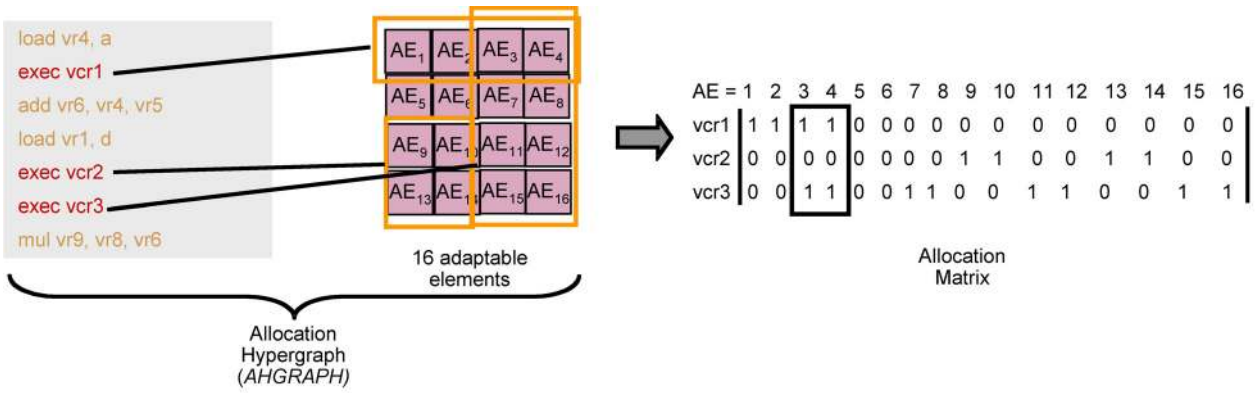

Figure 5: Hypergaph allocation strategy from configuration allocation is shown where custom instructions on are executed by *exec* instructions. The custom instructions are require a configuration that is mapped to a subset of computational resources denoted on the right-hand side of the figure.

The custom instruction scheduler is shown in Algorithm 2, and is based on greedy list scheduling. The *ResourcesAvailable* function is responsible for ensuring that resources are only assigned to the the current *cop* if and only if:

- No two custom instructions share AFU resources in the same cycle, and
- The total adaptive array resources used per cycle must be less than or equal to the total adaptive resources.

Returning to Figure 5, all of the custom instructions can not execute concurrently. This is because the custom instructions represented by *vcr*1 and *vcr*2 are both assigned to shared resources AE_3 and AE_4 by the allocation algorithm. As with configuration allocation, our scheduler adapts the traditional compiler infrastructure to solve this new problem of scheduling custom instructions.

Both our allocation and scheduling algorithms leverage existing EPIC compiler infrastructure which may translate into a more easily verifiable compiler and faster to market to target an AEPIC processor. These factors directly affect the productivity of the AEPIC design and development process.

4. RESULTS AND DISCUSSION

In this section, we present the experimental methodology and results. Our results are based on simulations of the AEPIC architecture, EPIC architecture and the DSP TMS320C6713-300 processor. We have chosen to implement our algorithms and simulator within the Trimaran [13] compilation framework, as shown in Figure 6. In our framework, we also utilize CoDeveloper from ImpulseC [1] as a configuration size and latency estimation tool to create the configuration library. We do not build the entire application in ImpulseC, but rather just the regions we want to use for custom instructions. The VHDL code was used for initial verification for the area requirements and latency of custom instructions, but the VHDL is was not used during simulation. Trimaran is used for all other compilation tasks, and for the simulation. The simulation executes the EPIC code and inserts the latency of the custom instructions when the processor encounters them. We have plans to integrate a fabric simulator in a future release, which will allow us to accurately model the frequenices and timing in the fabric. As a result, the outputs of the simulator are the number of cycles expended in various regions of the program and the total cycles executed. The DSP

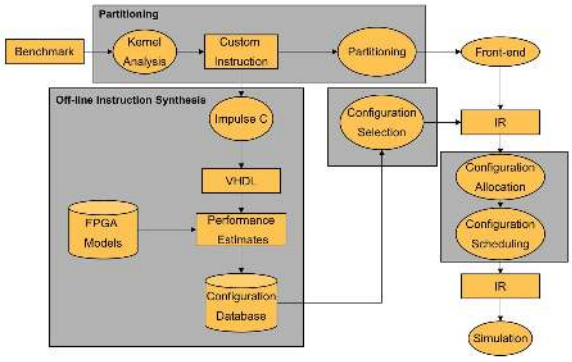

Figure 6: Experimental methodology that uses off-line partitioning, instruction synthesis, and selection coupled with automatic allocation, scheduling, and simulation

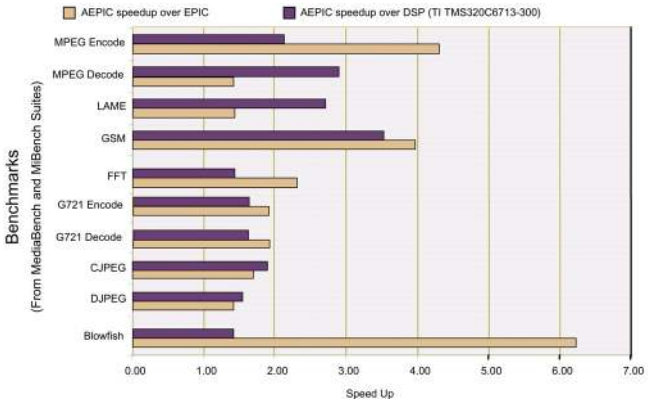

Figure 7: Speed ups of AEPIC compared to EPIC and TI DSP processor using embedded applications from the MediaBench and MiBench suites

is simulated using the Texas Instruments Code Composer Studio Development Tools [16] using the product compiler provided for it.

4.1 Main Results

In Figure 7 show the speed-ups that we achieved by applying out algorithms through the AEPIC architecture against the EPIC and DSP cases. These results are based on the following configurations of the simulators:

Trimaran *EPIC* simulator: Trimaran is equipped with one data-path, and thus it is configured with 4 integer ALUs and 4 floating-point ALUs along with 32 registers, and the same cache hierarchy.

Trimaran *AEPIC* simulator: Uses the same *EPIC* core simulator with additional support for executing the *AEPIC* ISA and custom instructions on the adaptive fabric. It supports configuration registers, array registers, and configuration cache hierarchy.

TMS320C6713-300 TI DSP: This DSP is equipped with 2 identical data-paths each with 1 fixed-point ALU, 2 fixed- and floating-point ALUs, and one multiplier. The DSP is also equipped with 16 registers for each data-path. In addition, there is a 4KB L1 cache and a 32KB L2 cache.

Our results show our algorithms consistently achieve gains over both of the competing processor designs. The performance improvements that we see range from 150% to over 600%. The average improvement across the benchmarks is about 200%. These improvements show the power our compile-time optimizations as they interplay with the *AEPIC* architecture. However, we were concerned with the area used by our design to achieve this performance. We equate the cost of performance with the chip area. This gives us a $cycle/mm^2$ cost that can be directly related to dollars in manufacturing and running the processor.

4.2 Comparing Hardware Costs

To understand the performance gains of our optimizations and architectural extensions through AEPIC especially as it relates to competing architectural styles, it is necessary to provide a mechanism to normalize our performance against the area that supports it.

	Itanium 2	**TI DSP**	**Virtex II Pro (XC2VP30)**	**AEPIC**
Functional Units	9	6	2	6
Caches	L1, L2, L3	L1, L2,	L1, L2	L1, L2
Area	$432mm^2$	$386mm^2$	$389mm^2$	$533mm^2$

Table 3: Architecture parameters affecting area

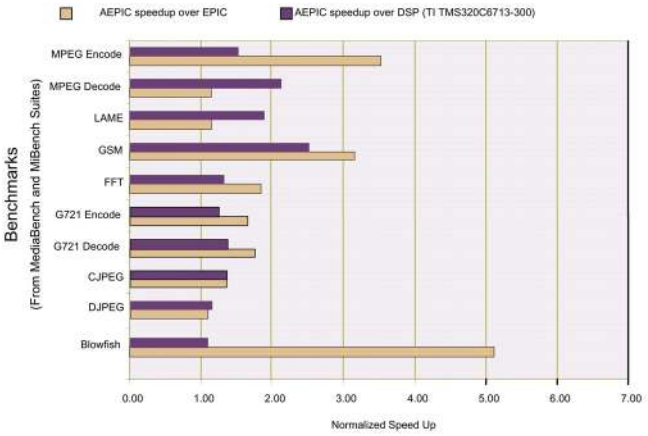

Figure 8: Speed-Ups normalized against area of AEPIC compared to EPIC and TI DSP processor

The area estimates we make here are based on the (limited) information publicly available about the details of the architectures we used in our study. Our attempt is to be conservative in our estimates such that our results in the AEPIC context are penalized rather than enhanced by the area estimates.

The Itanium [24] [20] [4], TI DSP TMS320C6713-300 [17], and Virtex II Pro [29] characteristics shown in Table 3 are based on respective published reports. The AEPIC reconfigurable array costs are based on those of the FPGA area from the Virtex II Pro. In addition, the area of the 6 EPIC functional units needed to be determined. Based on an inspection of the Itanium 2 [20], we determined that 66% of the area is devoted to the L3 cache, which is not a component of our AEPIC configuration. This leaves 33% for the processor core (includes registers, cache, and functional units), and thus $144mm^2$ of area for the EPIC processor of AEPIC. Therefore, our AEPIC processor is equivalent to $144mm^2 + 389mm^2 = 533mm^2$ in area.

In Figure 8, we show the results after normalizing the cycles against the area of the respective processor. As expected, these results show that the AEPIC execution model does provide a means to extract more performance from typical embedded system algorithms, where the gains now can be as high as a factor of 500% over an EPIC processor for the Blowfish benchmark, and over 250% in the context of the GSM benchmark over the TI DSP processor.

4.3 Exploring our Architectural Space

In conventional EPIC architectures, additional silicon area can be utilized to increase the number of functional units to implement the (fixed) ISA. This increase in the number of functional units should translate to an increase in performance for an application, till an architectural "sweet spot" is reached. Beyond this point, further investment in in area will yield diminishing returns in performance.

Area		GSM	
Adaptive Array Size	Number of Fus	Speed-Up(EPIC)	Speed-Up(DSP)
64	2	1.05	0.93
64	3	1.04	0.92
64	4	1.03	0.91
64	5	1.03	0.91
64	6	1.05	0.93
128	2	1.04	0.92
128	3	0.89	0.79
128	4	1.04	0.92
128	5	1.04	0.92
128	6	1.06	0.94
256	2	0.89	0.79
256	3	1.04	0.92
256	4	1.04	0.92
256	5	1.04	0.92
256	6	1.06	0.94
512	2	3.15	2.80
512	3	3.15	2.80
512	4	3.77	3.35
512	5	3.77	3.35
512	6	3.89	3.46
1024	2	3.14	2.79
1024	3	3.14	2.79
***1024**	**4**	**3.76**	**3.34**
1024	5	3.76	3.34
1024	6	3.89	3.45
2048	2	3.17	2.82
2048	3	3.17	2.82
2048	4	3.80	3.37
2048	5	3.80	3.37
2048	6	3.93	3.49

Table 4: Selected benchmark performance over range of architectural space parameters

Area		Blowfish		FFT	
Adaptive Array Size	Number of Fus	Speed-Up(EPIC)	Speed-Up(DSP)	Speed-Up(EPIC)	Speed-Up(DSP)
64	2	1.05	0.23	1.02	0.57
64	3	1.05	0.23	1.02	0.57
64	4	1.12	0.25	1.03	0.64
64	5	1.12	0.25	1.03	0.64
64	6	1.13	0.26	1.04	0.64
128	2	1.05	0.23	1.02	0.57
128	3	1.05	0.23	1.02	0.57
128	4	1.12	0.25	1.03	0.64
128	5	1.12	0.25	1.03	0.64
128	6	1.13	0.26	1.04	0.64
256	2	1.05	0.23	1.72	1.07
256	3	1.05	0.23	1.72	1.07
256	4	1.12	0.25	2.03	1.26
256	5	1.12	0.25	2.03	1.26
256	6	1.13	0.26	2.08	1.29
512	2	3.64	0.83	1.72	1.07
512	3	3.64	0.83	1.72	1.07
512	4	5.13	1.17	2.03	1.26
512	5	5.13	1.17	2.03	1.26
512	6	5.39	1.23	2.08	1.29
1024	2	3.64	0.83	1.72	1.07
1024	3	3.64	0.83	1.72	1.07
***1024**	**4**	**5.13**	**1.17**	**2.03**	**1.26**
1024	5	5.13	1.17	2.03	1.26
1024	6	5.39	1.23	2.08	1.29
2048	2	3.64	0.83	1.72	1.07
2048	3	3.64	0.83	1.72	1.07
2048	4	5.13	1.17	2.03	1.26
2048	5	5.13	1.17	2.03	1.26
2048	6	5.39	1.23	2.08	1.29

Table 5: Selected benchmark performance over range of architectural space parameters

The adaptive EPIC architecture introduces another dimension to this area-performance tradeoff, which is also true of any generic SoC. Specifically, in the AEPIC architecture, area can be used to increase the number of functional units, or alternately to realize the adaptive components. It is thus important to understand the impact of adding area to the AEPIC architecture, either through fixed functional units, or through increasing the size of the adaptive fabric—the goal is to identify the sweet spot.

This phenomenon is characterized in the Tables 4 and 5, where the optimal architectural configuration (in terms of number of EPIC functional units and the area of the adaptive fabric) for three representative benchmarks are summarized. All the benchmarks show increasing performance gains as area is increased with the sweet spot being reached when the adaptive component has an area that is comparable to 50% total capacity of a Virtex II FPGA, and the fixed EPIC component has 4 functional units. This is significant because we would not like to see increased area producing abberations in performance, and we would not expect that. More importantly, we have shown that with our current algorithms, the fabric can be reduced by 50% and still achieve our results. For our experiments, we have estimated that the total area required is $331mm^2$, which is significantly less area than the DSP and EPIC processors and achieves better performance.

We believe that more performance can be gained from global allocation and scheduling algorithms. This would allow us to determine when a configuration should not be unloaed because it will be used at a future point in the application outside of the current region. This would allow us to minimize the load/unload overhead for configurations. In addition, there are several applications with timing constraints, and this means that the processor must process the data at a target rate. A global scheduler would allow us to determine if timing constraints are being met by the application as a whole as the schedule and allocation mapping are being produced and changed during compilation.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a compiler-centric strategy for developing customized SoC solutions. Our work was done in the context of automating the development of custom hardware designs using compiler optimizations. We chose the hybrid architecture of AEPIC as a proof-of-concept because it provides the compiler with the necessary support for compiler-based customization. Our overall approach is to innovate techniques that perturb the standard optimizing compiler minimally. Thus, we aim to reuse large investments in program analysis such as live-ranges, interference graph construction, and scheduling frameworks with careful and *minimal*

extension to target an entirely new class of reconfigurable or adaptive SoCs. In this context, we do not generate code for a fixed ISA, but rather generate and coordinate customized hardware configurations.

We have shown that our compiler is able to automatically produce an allocation strategy for the configurations, and an execution schedule for the custom instructions. The novelty of our methodology is that we leverage existing compiler infrastructure in our modules for scheduling and allocation. This decreases the time and effort in developing our adaptive compiler from a traditional compiler. We have shown non-normalized speed-ups from 150% to over 600% and normalized speed-ups from 110% to over 500%. In addition, we identified an optimal architecture configuration, where cost in area is minimized and benefits are maximized. Our performance gains are attributable to several factors including highly optimized regions of the source program executing on the reconfigurable fabric, the ability of our allocator to select higher-performance configurations for allocation when resources are at a premium, and the ability of our scheduler to take advantage of available instruction-level parallelism. Finally, we put forth preliminary results capturing the productivity of our design process in the context of more traditional processes.

Our future work will investigate the phase-ordering problem of allocation and scheduling. This work will be done to determine which phase ordering is most effective. In addition, we will present other heuristics to more efficiently allocate and schedule custom instructions. The automation process will also continue as we develop automated techniques for partitioning and configuration selection, which will be based on integer linear programming and tree pattern matching. Finally, we will develop a set of metrics and a strategy for evaluating the productivity of hardware/software design processes.

In addition to the phase-ordering problem, we will be looking at global allocation and scheduling algorithms that enable us to schedule beyond function boundaries. This will address applications with global timing constraints and to globally minimize the loading/unloading of configurations.

Finally, we will examine the productivity aspects of hardware design and software design processes. Both industries guard their productivity data closely because of the proprietary nature of the industries. So, our analysis will, in all likelihood be based on empirical results, our experiences, and where possible, supporting documentation [5] [18].

When we compare ourselves to tools such as ImpulseC [1] and Celoxica [6], the latter cases include a synthesis phase in their flow that is consistent with the hardware design process. By contrast in the *AEPIC* context, the synthesis phase is not required because configurations are synthesized off-line and then the compilation-flow only has to make selection decisions. Thus, since synthesis is performed statically ahead of the application development and compilation process, in our approach, the cost of synthesis will be amortized over the development of several custom designs that may reuse the configurations, thus allowing significantly improved design productivity.

Acknowledgments

The authors would like to gratefully acknowledge the assistance provided by Romain Cledat, and Richard Copeland, Jr. of CREST at the Georgia Institute of Technology in providing the performance results for the TI TMS320C6713-300 processor, and Sung Kyu Lim for numerous helpful editorial comments.

6. REFERENCES

[1] I. accelerated technologies. http://www.impulsec.com/.

[2] Arc. http://www.arc.com.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[4] R. Belgard. Chart watch: Server processors. *Microprocessor Report*, 19(8):26–27, August 2005.

[5] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowits, R. Madachy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000. cocomo II.

[6] Celoxica. http://www.celoxica.com/.

[7] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocating via coloring. In *Computer Languages*, volume 6, pages 47–57, 1981. ACM Press.

[8] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, volume 17, pages 98–105. ACM Press, 1982.

[9] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. mei W. Hwu. Impact: an architectural framework for multiple-instruction-issue processors. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 266–275, New York, NY, USA, 1991. ACM Press.

[10] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *36th International Symposium of Microarchitectures*. IEEE Society, Dec 2003.

[11] E-Asic. http://www.easic.com.

[12] A. Excalibur. http://www.altera.com/literature/lit-exc.jsp.

[13] T. T. C. R. Framework. http://www.trimaran,org.

[14] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN symposium on Compiler contruction*, pages 11–16, Palo Alto, California, United States, 1986. ACM Press.

[15] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[16] T. Instruments. C6000 code composer studio development tools. http://dspvillage.ti.com/docs/catalog/devtools/.

[17] T. Instruments. TMS320C6000 technical brief. February 1999.

[18] H. H. Jones. How to slow the design cost spiral. In *Electronics Design Chain*. http://www.designchain.com/, 2002.

[19] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical report, HP Labs, Feb. 2000.

[20] K. Krewell. Best servers of 2004. *Microprocessor Report*, 19(1):24–27, January 2005.

[21] M. Kudlur, K. Fan, M. Chu, R. Ravindran, N. Clark, and S. Mahlke. FLASH: Foresighted latency-aware scheduling heuristic for processors with customized datapaths. In *International Symposium on Code Generation and Optimization*. ACM Press, 2003.

[22] L. Logic. LSI logic rapid chip platform ASIC:http://www.lsilogic.com/products/rapidchip_platform_asic/.

[23] S. Microelectronics. The Greenfield solution, http://www.st.com/.

[24] H. Packard. Inside the Intel Itanium 2 processor: an Itanium processor family member for balanced performance over a wide range of applications. July 2002.

[25] K. Palem and B. Simons. Scheduling time-critical instructions on risc machines. In *ACM Transactions on Programming Languages Systems*, volume 15, pages 632–658, 1993.

[26] K. Palem, S. Talla, and P. Devaney. Adaptive Explicitly Parallel Instruction computing. In *Proceedings of the 4th Australasian Computer Architecture Conference*, 1999.

[27] K. Palem, S. Talla, and W.Wong. Compiler optimizations for adaptive EPIC processors. In *Proceedings of the First International Workshop on Embedded Software*, October 2001.

[28] K. V. Palem, S. Talla, and W. Wong. Compiler optimizations for adaptive EPIC processors. In *Lecture Notes in Computer Science*. First International Workshop on Embedded Software, Springer-Verlag, Oct 2001.
This paper is a summary of Suren's PhD Thesis.

[29] X. V. I. Pro. http://www.xilinx.com/products/design_resources/proc_central/.

[30] M. Schlansker and B. R. Rau. EPIC: An architecture for instruction level processors. Technical Report HPL-111, HP Labs, February 200.

[31] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of non-programmable hardware accelerators. Technical report, HP Labs, 2000.

[32] B. So, P. C. Diniz, and M. W. Hall. Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In *Proceedings of the IEEE/ACM Design Automation Conference*. ACM Press, 2003.

[33] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design-space exploration in FPGA based systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002.

[34] S. D. P. C. Specs. http://www.extremetech.com/article2/0,1558,1639233,00.asp. chip specs.

[35] T. S. System. http://www.ics.uci.edu/ specc/.

[36] S. Talla. *Adaptive Explicitly Parallel Instruction Computing, PhD Thesis*. PhD thesis, New York University, 2000.

[37] Q. technology. http://www.qstech.com/.

[38] Tensilica. http://www.tensilica.com/.

[39] T. Way and L. L. Pollock. Evaluation of a region-based partial inlining algorithm for an ILP optimizing compiler. In *PDCS*, pages 698–705. International Society for Computers and their Applications, 2002.

[40] P. Yu and T. Mitra. Characterizing embedded applications for instruction set extensible processors. In *Proceedings of the Design Automation Conference*. IEEE Society, 2003.