Balancing Design Options with Sherpa

Timothy Sherwood[†] Mark Oskin[‡] Brad Calder[†]

sherwood@cs.ucsd.edu oskin@cs.washington.edu calder@cs.ucsd.edu

[†]Department of Computer Science and Engineering, University of California, San Diego

[‡]Department of Computer Science and Engineering, University of Washington

Abstract

Application specific processors offer the potential of rapidly designed logic specifically constructed to meet the performance and area demands of the task at hand. Recently, there have been several major projects that attempt to automate the process of transforming a predetermined processor configuration into a low level description for fabrication. These projects either leave the specification of the processor to the designer, which can be a significant engineering burden, or handle it in a fully automated fashion, which completely removes the designer from the loop.

In this paper we introduce a technique for guiding the design and optimization of application specific processors. The goal of the Sherpa design framework is to automate certain design tasks and provide early feedback to help the designer navigate their way through the architecture design space. Our approach is to decompose the overall problem of choosing an optimal architecture into a set of sub-problems that are, to the first order, independent. For each sub-problem, we create a model that relates performance to area. From this, we build a constraint system that can be solved using integer-linear programming techniques, and arrive at an ideal parameter selection for all architectural components. Our approach only takes a few minutes to explore the design space allowing the designer or compiler to see the potential benefits of optimizations rapidly. We show that the expected performance using our model correlates strongly to detailed pipeline simulations, and present results showing design tradeoffs for several different benchmarks.

1 Introduction

The continuing expansion of the embedded market has created a significant demand for low-cost high-performance computing solutions that can be realized with a minimum of engineering effort. While die area and power consumption is a concern when developing a desktop-processor, in the embedded design world these factors have a significant impact on the cost-point and ultimate functionality of a product. Application specific processors allow a designer to meet these goals by specializing the features of the processor to a particular embedded application. Architectural

choices such as cache sizes, issue width, and functional unit mix can be traded off in the cost-energy-performance space on a per-application basis. Customization may also include extensions in the form of custom accelerators, for instance the inclusion of encryption instructions [32]. Building embedded processors tuned to a specific application enables a design that minimizes area and power while still meeting performance requirements. At the same time, because the design is based on a carefully tested template, the design time is reduced and reliability increased significantly over a pure ASIC based approach.

In the desktop processing realm one might take an "everything and the kitchen sink" approach, focusing only on bottom-line performance. But in the fiercely competitive embedded domain system cost (area) is the dominating factor. Smaller processors means more integration resulting in less packaging cost, and less area means more chips per die and less defects per chip.

In this space, it is vital to determine how to effectively divide your area between the instruction cache, data cache, branch predictor, additional functional units, and other components. This is because cost is the single most important attribute for embedded applications. When designing a custom processor the designer needs to find the lowest cost implementation that meets the target performance constraints, and this problem by it's very nature requires that all optimizations be viewed in concert with all other parts of the processor. Faced with this extremely important goal of minimizing area and power, and a combinatorial number of design options, how should an embedded processor engineer proceed?

The goal of this paper is to introduce a technique that assists this processor engineer (or automated compiler) in the navigation of the vast application specific processor design space. It is not practical to evaluate every possible alternative through detailed simulation. But by building a set of estimators and solvers a very good characterization of the design space can be built and explored. We present the *Sherpa* framework as a guide for this application specific processor exploration. Sherpa efficiently explores the custom-processor design space because, while the space is non-linear and full of local minima it is, to first order, decomposable. Custom processors are currently built from in-order processing cores [13, 26, 4, 1], and the design features of these, as we will show in Section 5, can be broken apart and modeled as independent optimization problems.

Sherpa is a framework for exploring these sub-problems and recombining them to find ideal parameters for the usual processor features such as caches, and branch predictors. It is fully general, however, and can also explore

whether or not to use custom instruction accelerators, or variations on functional unit type, for instance whether to use a slow or fast multiplier, or no hardware multiplier at all. More significantly, it optimizes these selections against all of the other design possibilities.

The Sherpa framework begins with an application and carves up the entire processor design space into a set of loosely independent regions. Each region represents a logical architectural component, such as the data-cache hierarchy, branch predictor, custom accelerator, etc. These regions are then treated as separate sub-problems and are explored using data-driven analytical models or high-level simulation. The result of these independent explorations is a set of characterizing functions that are combined to form a model of the entire design space suitable for constrained minimization. This global model is then optimized through standard integer-linear programming techniques to arrive at desirable parameter selections for each architectural component.

While integer-linear programming has been widely used in the ASIC design community to automate dataflow graph partitioning, to our knowledge we are the first to apply this powerful optimization technique to total-processor design space exploration. The significant research advance we contribute in this paper is the characterization of the processor design space into component models, the constraint formalism to combine them, and the validation of these techniques in this environment. The research in this paper bridges the gap between optimizing small ASIC dataflow graphs, and complete processor designs; it lays the foundation for optimization of more complex architectures. Additional contributions of this paper are:

- An automated design space surveyor for embedded in-order custom-processors.
- A methodology to approximate various design options for processor features with piece-wise linear functions relating area to performance.
- An integer-linear programming formulation for combining these sub-problems using these piece-wise linear functions. This provides an approach for rapidly finding ideal parameters for the processor architecture.
- An in-depth analysis of trading off performance and area against each other. This leads to the result that when holistically designing a processor, instead of focusing only on a single component, area and performance optimizations can be used to the same effect.

- A demonstration of using Sherpa to decide whether to use a custom instruction accelerator. For our purposes we illustrate this with the design choice of whether or not to use a hardware multiplier, however, the technique generalizes to any custom instruction one is trying to decide whether or not to include in a design, and several different alternatives can be evaluated at the same time.

- A demonstration of how to use Sherpa to quickly evaluate the potential benefit of an architectural optimization. As a case study we examine the use of application specific finite state machine predictors for branch prediction [25].

2 Design Navigation

There are currently several commercial ventures that offer customizable RISC processors [13, 26, 4] such as the XTensa processor from Tensilica [13]. Tensilica provides support for XTensa in terms of compilation, synthesis, and verification [21]. Embedded designers use these tools by providing a high level specification for a processor core. Produced from them is a component ready for integration onto a systems-on-a-chip (SOC) product. While this is a significant advancement for customized processors, the designer is still for the most part unassisted in determining the optimal processor specification for their embedded application. This is where our research is focused with the Sherpa framework.

In this section we discuss the typical design flow used when generating a customized processor. We then illustrate where Sherpa integrates into this workflow to provide rapid feedback and design optimization.

2.1 Process Flow for Creating an Application Specific Processor

We will discuss the workflow used to create an application specific processor, and then how Sherpa enhances it. For concreteness our discussion will focus on the process used for systems similar to the XTensa processor from Tensilica, however, Sherpa can also be used by more automated tool-chains such as those proposed for the PICO [1, 2] custom processor (discussed further in Section 6). The current design process, as follows, involves several iterated steps to find an optimized processor configuration:

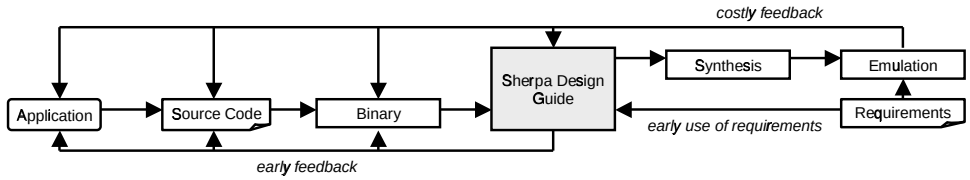

Figure 1: The Application Specific Processor design flow with the Sherpa design guide. Sherpa minimizes the number of times the design must be iterated on. This allows the designer to spend their time optimizing the algorithms used or the way they are coded instead of having to iterate over the design space.

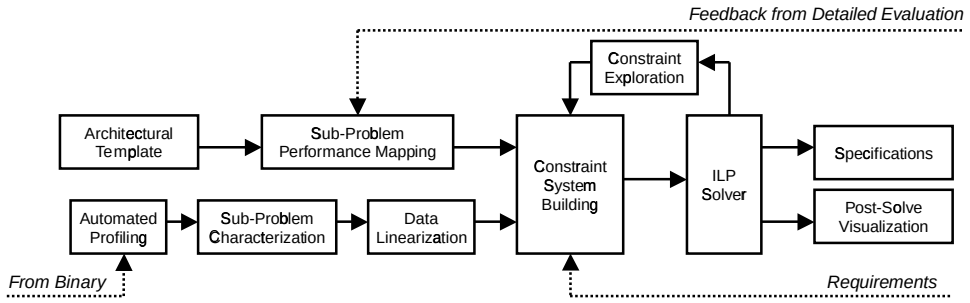

Figure 2: Overview of the Sherpa design guide implementation. The system starts with the input binary and the set of requirements. The binary is profiled and the different sub-problems are analyzed. In the data linearization stage a set of piecewise linear functions are built for each sub-problem. These, along with an automatically constructed performance model, are converted into a constraint system which is solved using integer-linear programming. A set of different constraints can then be iterated over. The output of the solver can then be visualized for human examination and converted into a set of processor specifications for use by synthesis.

1. The embedded application is provided in a high-level source language, such as C or Java. The design team determines performance targets, e.g. page renders per minute, or frames per second.

2. The application is compiled to a binary, or in some cases several different binaries depending upon the architectural options (available registers, instructions, etc).

3. Profiles of the application on the target platform are generated from simulation. These profiles focus the design teams effort on application and architectural bottlenecks. Architectural changes involve a lengthy, manual enumeration of the potential options such as cache and branch table sizes in an attempt to find an ideal configuration.

4. The design team then creates a new binary compiled to the potential architectural choices from the previous step. The processor specification is passed through the backend toolset, which generates an RTL description

of the custom processor.

5. The binary is then re-simulated on the new custom processor and the entire process is iterated to further optimize the configuration.

The above process, when done manually, can be tedious and time consuming. It leaves to the designer the choice of what architectural options to modify by guessing the possible outcome prior to executing a lengthy tool-chain and simulation step. Unfortunately, the entire space of design options for a configurable core is vast in both performance and cost. Furthermore, this search space is filled with local optimas requiring something more clever than a straightforward greedy approach.

2.2 Using Sherpa to Guide Design

While engineers can perform the above process by hand, they need to first spend significant time becoming familiar with the target application, and then must quantify the behavior of the program through exhaustive experimentation and analysis. We instead provide Sherpa as an automated system to examine program behavior and suggest near-optimal (which are often optimal) design decisions, allowing the designer to make informed decisions early and experiment with different optimizations. In addition, Sherpa also allows the designer to more easily deal with processor design tradeoffs resulting from late changes to the application.

Figure 1 shows the application specific design flow when using the Sherpa design guide. The goal of the Sherpa design navigator is to allow its user to make design tradeoffs early in the exploration phase and quickly narrow in on the bottlenecks of a system. Sherpa analyzes the application code from the existing custom processor design workflow and quickly searches the entire architectural design space for global optima. The embedded product design team then uses this information to narrow in on the final custom processor architecture.

Figure 2 shows the internal process of the Sherpa Design Guide (the gray box) shown in Figure 1. Sherpa takes the input application, and profiles it to generate a set of representative traces. Next, Sherpa decomposes the vast custom processor design space into a set of independent sub-problems (e.g., data cache, branch predictor, etc). The application trace is then used to initiate these models. Finally, Sherpa combines the models using an integer-linear program (ILP) solver to locate a globally ideal custom processor architecture configuration. This configuration,

along with the design tradeoffs Sherpa made, is then passed back to the designer for examination. The designer can then focus attention on the significant architectural components and application sections.

3 Implementation

We now describe the implementation of a Design Navigator, Sherpa, beginning with a description of our overall approach and a description of the target architecture, and finishing with a detailed evaluation of the algorithms used. We use the design navigator to find ideal sizings for instruction and data cache, branch prediction logic, and multiplier settings.

3.1 Overall Approach

Our initial implementation is based on a constraint system that expresses the entire range of architectural options. The constraint system is structured so that a integer-linear solver will find an optimal solution from the possible design options while maintaining hard constraints, such as performance needs, but at the same time optimizing for design goals (such as area).

Our approach initially treats each customizable component separately. For each component, an area-performance model is developed by simulating a variety of component configurations with a representative instruction trace. This collection of area-performance models is then combined with integer-linear programming and solved to yield an ideal overall processor configuration. Before discussing the details of this process we wish to describe the customizable processor we are targeting in more detail.

This paper optimizes a customizable architecture similar to the in-order XTensa Processor [13]. The reason that we have chosen to use this processor model is that the work done by Tensilica is mature enough that they have a workable design path from processor specification all the way down to silicon. They are able to do this because they work with the CAD tool designs further down the design chain to insure that the parameterized processors they build will be able to be synthesized into an efficient form. The Sherpa guide can be used as an automated front end to the XTensa design process to guide the selection of ideal parameters of the different components.

The parameterized processor we model is a single issue in-order 32-bit RISC core with parameterizable data

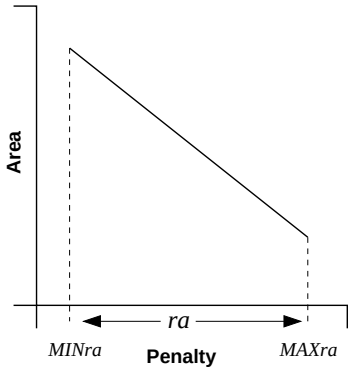

Figure 3: This graph shows two design points MINra and MAXra, and the creation of a simple linear function between these two points relating performance to area.

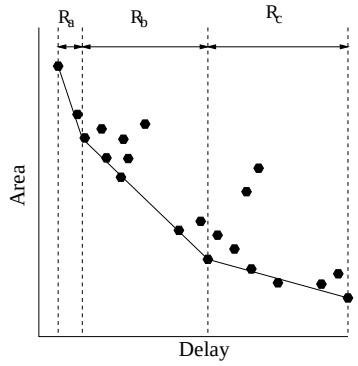

Figure 4: A graphical representation of the Piecewise Linear Model which is used to quantify the area-performance tradeoffs for each sub-problem. The Pareto optimal design points are estimated by intersecting line segments for which the slope and intercept are known.

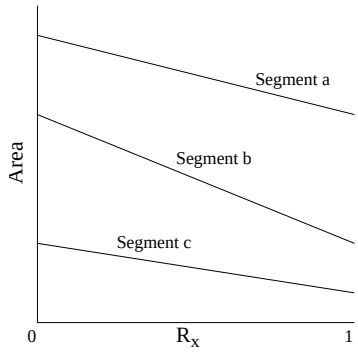

Figure 5: Translated piece-wise linear model. Note that for each line segment we see the same height on the y-axis but now the lines are stretched to be over the range of $[0, 1]$.

cache, instruction cache, branch prediction, an optional fast integer multiplier and core. There is a basic core size which contains the primary functionality of the chip such as the execution box, instruction handling, bus controller, and memory management unit. We assume that the area of this core is unchanging and leave its optimization to future work. The parameterizable parts of the processor, are more auxiliary components used to reduce delay.

3.2 Simple Linear Constraint System

Suppose we wish to customize only a single architecture parameter; for example the instruction cache size. Then we could simply model the performance of the processor with various cache sizes and pick the smallest one that still met our performance requirement. However, suppose instead that we were asked to choose the ideal instruction and data cache sizes that together made a system that met the performance requirement. In our case we define "ideal" as the configuration with the lowest area. A naive approach is to enumerate through and simulate all possible instruction and data cache size combinations and choose the optimal one. This exhaustive search will find the optimal configuration but is neither fast nor scalable.

Our approach is to divide this design space into two sub-problems (instruction cache area versus delay and data cache area versus delay), and to explore each sub-problem independently. The model presented in this paper

approximates the performance penalty from a sub-problem as a fixed length processor stall, where this stall affects the entire pipeline. For example, a cache miss can be represented as a fixed 50 cycle processor stall, and a branch misprediction as a 6 cycle stall. Even though all of the sub-problems we examine are not completely independent (e.g., branch prediction and instruction cache delays) the probability of these delays overlapping for an in-order processor is small for the programs we examined. We measured the amount of dependence between the different subproblems using detailed cycle accurate simulation and found it to be negligible. This issue is discussed further in Section 5.

By allowing ourselves the simplifying assumption that these are independent sub-problems we can express the overall performance of our example two parameter processor by Equation 1:

$$T_{Total} = T_{Base} + T_{DataStall} + T_{InstrStall} \tag{1}$$

where T_{Total} is the total execution time of the program, T_{Base} is the time to execute the program assuming a perfect memory, and the stalls for both the data cache and instruction cache are added for the corresponding configurations being considered.

In order to compare the different configurations, we need to create a function that relates area to performance. Figure 3 shows the performance versus area for two cache configurations chosen for the data cache. For purposes of illustration, we assume this is a simple linear function; the next section introduces more realistic models. Given the two points in Figure 3, the contribution to processor stall from the cache can be expressed in terms of its structure via the equation $T_{Stall} = k \times Area$, where k is a constant. Performing this operation on both sub-problems creates two independent linear functions that relate performance to area that must be combined to find the ideal balance between the two. Rewriting Equation 1 we get:

$$T_{Total} = T_{Base} + (k_{Data} \times Area_{Data}) + (k_{Instr} \times Area_{Instr}) \tag{2}$$

Using this equation we set a performance constraint $T_{Total} \leq T_{max}$, and solve for the two unknowns: $Area_{Data}$

and $Area_{Instr}$. The goal is to maintain this performance constraint while minimizing the area, which is expressed as:

$$Minimize : Area_{Total} = Area_{Data} + Area_{Instr} \tag{3}$$

The result of this formulation is a system of linear equations and an optimization criteria. This can be effectively solved using linear programming tools. Commercial solvers from CPLEX are available, but we found the freely available *lpsolve* tool to be sufficient [6].

3.3 Data Linearization

The previous example of finding the ideal balance between a set of instruction and data cache configurations relied upon the assumption that there is a strict linear equation to represent cache area to performance. Clearly this assumption is simplistic and we must support a more complex area-performance model.

For this we turn to a piece-wise linear approximation of the actual function. To form this approximation we begin by identifying Pareto-optimal design points. A point p is said to be Pareto-optimal (on the axis in Figure 4) if there is no point p' such that p' requires less area and incurs less stall time. Intuitively, if a point is not Pareto-optimal then there is another design point that will achieve as good or better performance with less area. We use these Pareto-optimal points to help form a piece-wise linear approximation of the area-performance function.

Starting from the Pareto-optimal design points we greedily construct straight line segments. A point is added to a line segment if the correlation between that line segment and all the points that approximate it does not drop below a threshold. The correlation coefficient, r, provides the extent to which the bounding Pareto-optimal points being considered lie on a straight line. For the results we present in this paper we insure that r is always greater than 0.98. When the correlation coefficient drops below this threshold, a new line segment is started. To insure that the line segments always intersect, the last data point included in a given segment is also included in the next segment.

The result of piecewise linearization can be seen in Figure 4. The points that are Pareto-optimal are connected by line segments that capture the overall area-performance trend. In this example three line segments R_a, R_b and R_c

make up the entire piece-wise linear approximation. Our goal now is going to be to modify the constraint system to use these far more precise sub-problem models, which we describe next.

3.4 Final Constraint System

A mixed integer-linear program is similar to a linear program except that some variables are optionally constrained to be whole numbers. Solving a mixed integer-linear program as opposed to a linear program is a straightforward process. We refer the interested reader to [17] for details on the internals of this algorithm.

To create our mixed integer-linear constraint system, the first step is to include each piece-wise linear function for each sub-problem into the constraint system. This is performed by breaking up each line-segment of the piece-wise linear approximations into separate components. These line segments are then translated to have a uniform length and starting point along the horizontal axis. Figure 5 depicts the same piece-wise linear function from Figure 4 after translation is performed. For each piece, we generate the functions for delay and area where the function for area and delay has been normalized between zero and one.

The linear function for area is calculated as

$Area_x = BaseArea_x + SlopeArea_x * R_x$, and the function for delay is calculated as $Time_x = BaseTime_x + SlopeTime_x * R_x$. In both of these equations, R_x is a design point with a value between zero and one. $BaseArea_x$ and $BaseTime_x$ are the start area and time for each linear piece in Figure 4.

Once distilled into separate line-segments we construct a mixed integer-linear program where only a single line-segment (from the piece-wise linear function) is selected for each sub-problem. The single line-segment is chosen using integer programming. To illustrate, Figure 6 depicts a new constraint system with these constraints. This constraint system works by choosing a tuple (iR_x, R_x) with Equation 4. iR_x is an integer value of either zero or one. R_x is a linear value between zero and one. This tuple selects which line segment to use with the integer variable iR_x, and the design point within that line-segment to examine using the variable R_x. Equation 4 cannot be implemented directly, and we describe next how to construct it using integer and linear constraints.

Equations 5 and 6 perform a line segment selection for the sub-problem by ensuring that only one line segment is selected, since each iR_x can only have a value of 0 or 1. Equation 5 insures that only one iR_x will be equal to one, and all the rest will be equal to zero. Since we normalized the linear functions to represent the area and delay

$$(iR_x, R_x) = ChooseOne\left(\{(iR_1, R_1), (iR_2, R_2), ..., (iR_n, R_n)\}\right) \tag{4}$$

$$iR_1 + iR_2 + ... + iR_n = 1 \tag{5}$$
$$iR_1 \geq 0,\ iR_2 \geq 0,\ \ ...\ ,\ \ iR_n \geq 0 \tag{6}$$

$$0 \leq \quad R_1 \quad \leq iR_1 \tag{7}$$
$$0 \leq \quad R_2 \quad \leq iR_2 \tag{8}$$
$$\quad\quad ...$$
$$0 \leq \quad R_n \quad \leq iR_n \tag{9}$$

$$\begin{aligned} 0 \leq \quad & Time_{required} \ - \ Time_{perfect} \\ - \quad & (BaseTime_1 * iR_1 + SlopeTime_1 * R_1) \\ - \quad & (BaseTime_2 * iR_2 + SlopeTime_2 * R_2) \\ - \quad & ... \\ - \quad & (BaseTime_n * iR_n + SlopeTime_n * R_n) \end{aligned} \tag{10}$$

$$\begin{aligned} Minimize: \quad & (BaseArea_1 * iR_1 + SlopeArea_1 * R_1) \\ + \quad & (BaseArea_2 * iR_2 + SlopeArea_2 * R_2) \\ + \quad & ... \\ + \quad & (BaseArea_n * iR_n + SlopeArea_n * R_n) \end{aligned} \tag{11}$$

Figure 6: Constraint system for a piece-wise linear sub-subproblem model.

using a variable between 0 and 1, we are able to reuse the variable iR_x to subtly constrain the selection of the design point R_x. This is shown in Equations 7 - 9, where the design points along the chosen line segment are examined.

Once a given tuple is selected, the translated $Base_{area,time}$ and $Slope_{area,time}$ values are used to determine performance impact and area cost of the choice. The modified performance bound is illustrated in Equation 10 and the new minimization goal is shown in Equation 11.

Figure 6 shows only the constraint system for one sub-problem. To generate the overall constraint system to examine the trade-offs between multiple sub-problems, each sub-problem has its own constraints to choose an iR_x and R_x particular to that sub-problem using equations similar to those shown in Equations 5 - 9. The time delay for each sub-problem is subtracted from Equation 10, and the area delay for each sub-problem is added into Equation 11. Then the overall system is run through the constraint solver to minimize the area, while meeting the

specified $Time_{required}$.

4 Sub-problem Characterization

In the previous sections we presented our general methodology of partitioning the complete application specific processor design space into a set of loosely independent sub-problems. In this section we discuss the sub-problems we explored and how we modeled them. The sub-problems of the design space that we chose to explore in depth are the instruction and data caches, the branch predictor configuration, and whether or not to include a fast hardware multiplier. The general methodology for constructing each of these models is to explore a sampling of points through direct simulation and then construct piece-wise linear models from the Pareto optimal design points.

For a given subproblem (e.g., branch predictor) we can trade off multiple designs (gshare, bimodal, 2-level) by plotting them all on the same Pareto graph. The piece-wise linear model built from that graph represents the ideal design to use for a given performance/area point. The same approach would be taken if we were to have the option of different types of accelerators for a given instruction or sets of instructions.

In this section we describe how we estimate the area for each design point considered as well as how to calculate the performance penalty for that configuration for the trace being examined.

4.1 Data and Instruction Cache

We start with an examination of the caches in our architecture. As will be seen later in Section 5.2, the caches are the dominant configurable area of an application specific processor. Because of this it is imperative that the models correctly capture the area-performance tradeoffs since they are the first order determinant of the overall accuracy of our technique.

Cache design is a well-studied problem in computer architecture, and we wish to build upon past work in this area where appropriate. Reinman and Jouppi [23] present a very detailed cache design space walker based on the work presented in [31]. CACTI, when given a set of cache parameters, such as associativity and cache size, will find the delay optimal cache array partitioning scheme (i.e. the fasted physical device layout for the given cache parameters). Mulder [20] presents a validated area model for caches and register files. For our research we use the

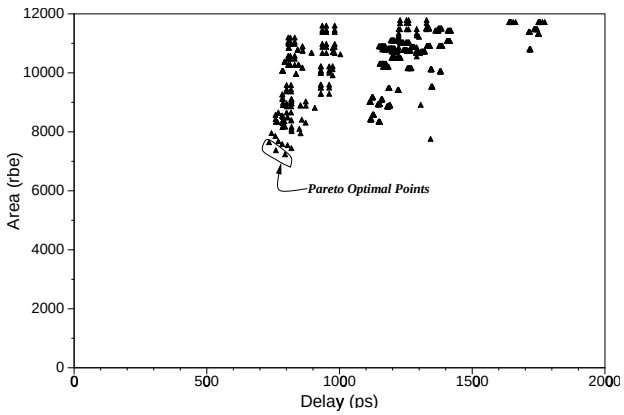

Figure 7: Graph of area-delay tradeoffs for a single cache configuration. This graph was generated by modifying the CACTI tool to output info on all configurations considered. For the example cache parameter set shown, over 9000 different configurations were considered but only three Pareto optimal configurations were found.

CACTI design tool to aggressively optimize the partition of the cache and then use that configuration to calculate the area using Mulder's area model.

One concern with using CACTI to optimize for area-optimal caches is that its internal optimization goal is programmed to find delay-optimal designs. To verify this would not skew our results we modified the CACTI tool to produce Figure 7, which depicts the delay versus area tradeoff of over 9,000 cache partitions that CACTI examined for a single cache parameter set. Out of all of these configurations examined by CACTI, only three Pareto optimal ones were generated. This shows that the most performance-efficient partition is extremely close to the most area-efficient. Figure 8 depicts this area variation for a range of different cache configurations. We found at most a 4% difference in area from the most area-efficient to the most performance-efficient design. Therefore, we chose to use the delay optimized cache partitioning from CACTI for our results. Fortunately the cache partitionings are not program dependent hence they can be calculated once and stored in a database for later use by any optimization process.

To capture the program-dependent effect of the different cache options, we need to estimate the number of cache hits and misses the target applications will have for each cache configuration. This can be done through simulation or analytical modeling [3]. Efficient cache simulators have been proposed to simulate many different configurations at once [29]. For our work we found that we could quickly simulate all of the reasonable power of two cache sizes

Size/Assoc	1	2	4	8
1K	±2.68%	±3.71%	±0.23%	±0.00%
4K	±3.30%	±0.49%	±0.25%	±0.12%
16K	±3.56%	±1.32%	±0.79%	±0.13%
64K	±2.14%	±0.65%	±0.66%	±0.40%

Figure 8: Percent area variation between the most area efficient design examined by the CACTI design tool, and most delay efficient design. Results are shown for a 1K to 64K cache from direct mapped to 8-way associative. In CACTI, the cache partitioning scheme with the minimum delay is returned. A variation of less than 4% is seen in area when optimizing a cache configuration for area versus performance.

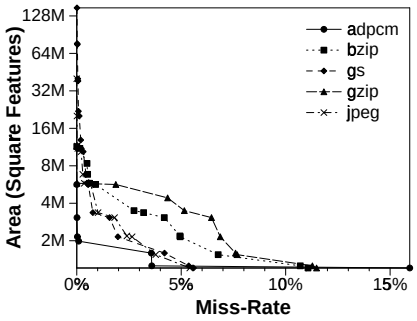

Figure 9: Instruction cache miss rate versus area tradeoff for different cache sizes, associativities, and line sizes. Only the Pareto Optimal points are shown.

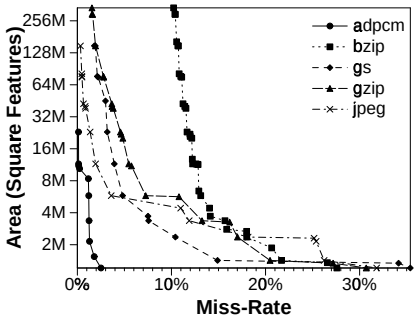

Figure 10: Data cache miss rate versus area tradeoff for different cache sizes, associativities, and line sizes. Only the Pareto Optimal points are shown.

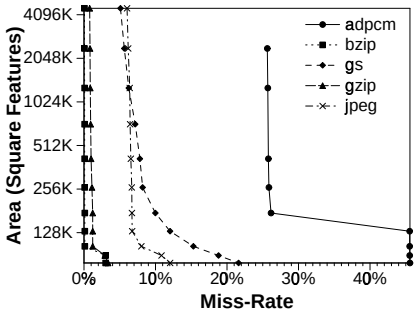

Figure 11: Branch misprediction rate versus area tradeoff. Several branch prediction architectures were examined and the Pareto optimal area/miss rate points are plotted for each program.

(256 bytes to 64K), associativity (direct mapped to 4-way and fully associative) and line size (8 bytes to 64 bytes) options. We then find the area of each of these configurations using the previously mentioned area model and CACTI partitioner.

Figure 9 shows the area versus miss rate for the instruction cache for the applications we explored, and Figure 10 shows the area versus miss rate for the data cache. From these points, we created the piece-wise linear models as described in the prior section.

Since our application specific processor configuration is for embedded in-order processors, we can easily map the cache miss penalty to performance for our constraint system. We assume that each cache miss results in a penalty of 50 cycles, and the processor stalls on a cache miss. We also examined hit under miss data caches, but that optimization provided only marginal performance gains. Therefore, in this paper we only present results for caches that stall on misses.

4.2 Branch Predictors

To examine the custom design space for branch prediction we examine selecting from the following well know branch predictors: (1) a table of 2-bit counters, (2) a global correlation predictor, and (3) a meta predictor that uses both local and global correlation information.

We examine the branch prediction miss rate for these different predictors for a variety of table sizes. Figure 11 shows the area versus miss rate for the Pareto optimal points in the different programs examined. For all of the programs examined the most area efficient branch predictor design was the table of 2-bit predictors similar to the one used in the XScale embedded processor [18]. To achieve better performance, significant resources are needed by the other predictor methods. For example, the Pareto curve for `adpcm` chooses a per-branch 2-bit for the low area points, and a meta predictor for higher area. It shows that the per-branch 2-bit counters give the lowest miss rate for an area less than 128K square features, and a meta predictor gave the best miss rate for an area greater than 128K. In crossing this design boundary the area is increased above 128K, but the misprediction rate is reduced from 45% down to 26%.

The misprediction rate measured here is used to calculate the performance penalty for our in-order processor model. By assuming a 6 cycle stall for each branch misprediction, and knowing how many branches we mispredict, we can estimate the total number of stall cycles incurred by the branch predictor.

4.3 Multiplier Unit

The last configurable option we examined was a processor with and without a large fast hardware multiplier instead of the much smaller and slower iterative multiplier. This is modeled by counting the number of multiply instructions executed in each program. The reason that we have chosen this configurable option is to show that our design optimization technique can also cleanly handle binary decisions such as whether or not to include a specialized functional unit. While deciding whether or not to use a fast multiplier is a binary decision, in an overall context the decision is complicated by the fact that its usefulness must be traded off against other, non-binary decisions. Thus, the actual decision is extremely difficult to answer by conventional techniques.

We model a hardware multiplier in the following way: if the fast hardware multiplier is used, then a multiply

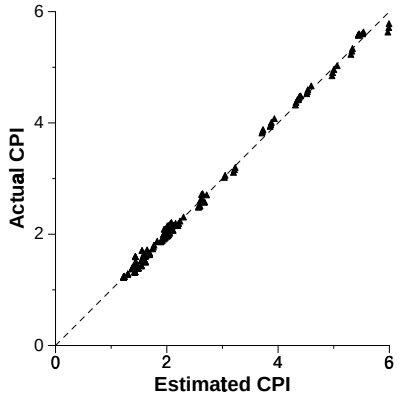

Figure 12: Verification of Performance Estimator. The estimated CPI time as calculated by our formulation is plotted as a function of the performance determined through detailed cycle level simulation.

takes 2 cycles, with the area cost of 3 million square features. The area of the multiplier is derived from [24]. If the fast hardware multiplier is not used, the multiply is performed with a software routine, estimated to take 250 cycles to execute [27].

4.4 Core Area

Finally we are left with the area of the actual execution core. Inside the core is the data path of the machine along with all of the control. While the control and data paths could be further optimized, we leave these subproblems for future work. These include the instruction fetch control, the instruction memory management unit, the data memory management unit, the bus controller and the basic functional units. The area for the remaining non-configurable set of functionality we used for this research is derived from [24] and is estimated to be 21 million square features.

5 Putting It All Together

Now that we have seen how the constraint system is built and solved in Section 3 and how the sub-problems are formulated and characterized in Section 4, we are ready to examine how they work together to explore the design space.

5.1 Running the Overall Solution

When running the system on a given program we must go through a few steps. The first step is sub-problem characterization as presented in the previous section. During this step we perform simulation of the different design options for each sub-problem. From this we generate the Pareto-optimal points and piece-wise linear approximation to be used for the next stage. The simulation time here varies both on what is being investigated and the program itself. We use fairly brute force methods for examining each space, and this process is the most time consuming of all the operations.

We built our simulation infrastructure on top of ATOM [28] because it provides both high performance and ease of use for a RISC architecture similar to those supported by configurable cores. The design point enumerations for each of the subproblems are calculated at run-time. For the programs we examined it takes anywhere from a couple of seconds to 10 minutes per simulation. However this process can be easily speed up though the use of smarter simulation algorithms [29], or analytical modeling as described in Section 6.3.

In addition to generating the estimated performances of the different sub-problems, we also need to estimate the total area consumed by each design point. This is done once for each given subproblem, and the results can be used for each of the different programs. This step takes less than 1 minute to run.

The final step of the optimization process is the actual construction of the constraint system. This step is very fast since many different design parameters can be examined quickly. To create charts of the pareto-optimal solutions, shown in Figures 13 and 17, we generated almost 50 design solutions in less than 8 seconds. Each one of these design solutions is the ideal combination of the different sub-problems and represents the best design point of many hundreds of millions. Because of this fast turn around many new design choices can be evaluated in real-time. For example, one could answer the question: what would the area impact be of reducing the cache miss penalty by 30%. This could be answered without re-running any simulations.

5.2 Results

We used ATOM [28] to profile the applications, generate traces and simulate cache and branch models. All applications were compiled on a DEC Alpha AXP-21264 architecture using the DEC C compiler under OSF/1 V4 operating

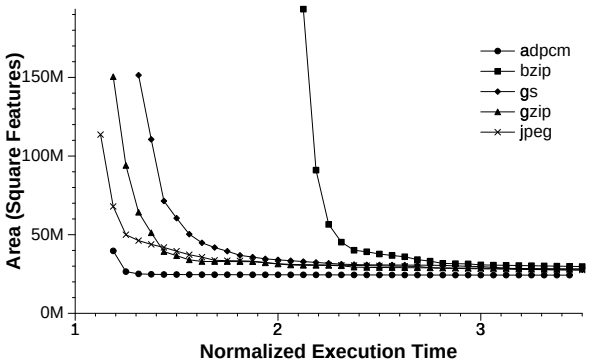

Figure 13: Total Area (in millions of square features) as a function of Normalized Execution Time for the different programs. The areas shown represent the minimum total size of the optimized core that can achieve the performance shown. The execution time is normalized to the performance of the base executable executed with no stalls of any kind.

system using full optimizations (`-O4`). We chose a set of five benchmarks which could have applications in an embedded environment The application `ijpeg` is from the SPEC95 benchmark suite, and `gzip` and `bzip` are from the SPEC2000 benchmark suite. We also include two programs from the MediaBench application suite – `adpcm` is a speech compression and decompression program, and `gs` is an implementation of a postscript interpreter.

The first step in testing our system is to verify that the performance we estimate using our combination of piecewise linear subproblems accurately matches with a detailed pipeline simulation of the hardware. We compare the estimated CPI gathered for several design points chosen at random, with the CPI of a detailed cycle-level simulation using SimpleScalar 3.0a [7]. We assumed in our constraint model and the simulator that all cache and branch structures could be accessed in one cycle. The processor was single issue, and used the same latencies for the different sub-problems as described in section 4.

Figure 12 shows the results of this verification procedure. In this graph we have plotted the estimated performance of the processor from our constraint system against that found through detailed cycle-level simulation. The results show a strong correlation between the estimates and actual values, with a correlation coefficient of $r = 0.99829$ for over 200 configurations chosen at random for evaluation across the different benchmarks. This shows that our performance estimation is accurate (otherwise the points in Figure 12 would show trends that do not follow the diagonal line drawn on the graph.)

Table 1 shows our results to further verify the independence of subproblems. The first column in the table,

Fixed	Vary	Avg StdDev	Max StdDev
D-cache	I-cache	0.000000%	0.000000%
D-cache	Branch	0.000000%	0.000000%
I-cache	D-cache	0.004775%	0.046030%
I-cache	Branch	0.001985%	0.038971%
Branch	D-cache	0.007916%	0.080000%
Branch	I-cache	0.006044%	0.095263%

Table 1: Quantifying the independence of subproblems. The first column in the table, labeled *Fixed*, shows the subproblem under examination, and the second column, labeled *Vary*, is the subproblem to which independence is being evaluated. For example, the first row in the table compares the independence of the data cache miss rate from the instruction cache miss rate. For the first row this is determined by holding the data cache size constant, varying the size of the instruction cache, and analyzing the change in data cache miss rate. This is then repeated for several sizes of data cache. The numbers reported are the average standard deviation and maximum standard deviation in miss rates across all fixed sizes evaluated.

labeled *Fixed*, shows the subproblem under examination, and the second column, labeled *Vary*, is the subproblem to which independence is being evaluated. For example, the first row in the table compares the independence of the data cache miss rate from the instruction cache miss rate. For the first row this is determined by holding the data cache size constant, varying the size of the instruction cache, and analyzing the change in data cache miss rate. This is then repeated for several sizes of data cache. The numbers reported are the average standard deviation and maximum standard deviation in miss rates across all fixed sizes evaluated. The results show that the subproblems are independent with less than a 0.1% standard deviation at maximum when holding one component constant and varying the other component. It can therefore be concluded from this graph that the data cache miss rate does not change significantly as the instruction cache miss rate varies for the cache sizes and processor model we are exploring.

We now examine the results of running the entire system on several programs. Figure 13 shows the resulting minimized area for each program for several performance design points. The area is shown in square features. The performance constraint is shown normalized to the base executable executed with no stalls for any components. For example, in order for `gs` to be executed in 1.5x the amount of time it would take to execute it with no stalls of any sort, the processor will need an area of 60M (square features).

As expected, relaxing the performance constraint reduces the area needed by the processor. All of the programs exhibit a very strong elbow in their performance area plot. Hence, the major working sets of the program are captured

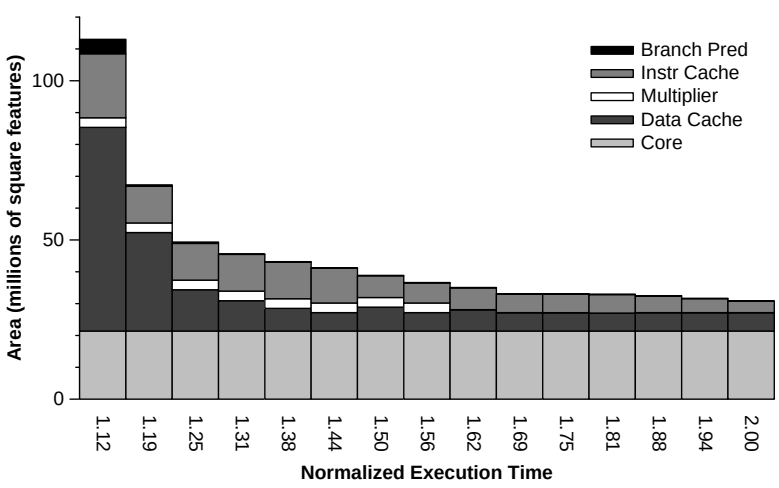

Figure 14: A zoom in of the tradeoffs made for the application `jpeg`. The axis are the same as shown in Figure 13 but here only a small portion of the total constraints examined are shown. In addition, the area of each component has been broken out into a separate stack to show the relative importance of each sub-problem for this application.

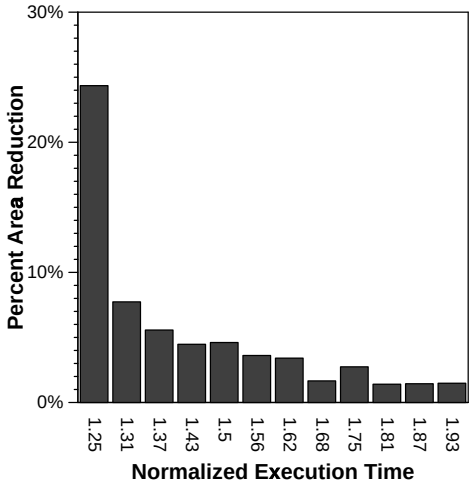

Figure 15: Percent reduction in area for `gs` when adding the option to use a custom finite state machine predictor for branch prediction into our Sherpa constraint system.

using a small amount of resources, and you will have to add significantly more more area on top in order to improve performance.

A closer look at an individual application shows how the constraint system trades off performance and area between all of the different sub-problems we examined. Figure 14 shows this breakdown for the program `ijpeg`. When the performance is tightly constrained, a great deal of area is devoted to the cache and branch predictor, as well as including the special integer multiply functional unit. As we relax the performance constraint, the optimizer balances off the different sub-problems in such a way that the performance criteria is met and the area is minimized.

In relaxing the performance constraint, the first area component to be reduced is the large branch predictor, while both the data and instruction cache are significantly reduced in size. Over the performance range between 1.14 and 1.44 the instruction cache size does not change significantly and the data cache is reduced.

It is interesting to note that analysis of our design tradeoff graphs can be be used to find the working set sizes for the caches. Between 1.44 and 1.50 the performance constraint is relaxed to the point that the instruction cache can be shrunk to the next working set size. However to make up for this increase in instruction miss rate, the data cache actually has to increase in size. This sort of tradeoff can again be seen between 1.56 and 1.62 where the multiplier is cut out and the size of the data cache is again increased. It is these sorts of complex tradeoffs that our system has

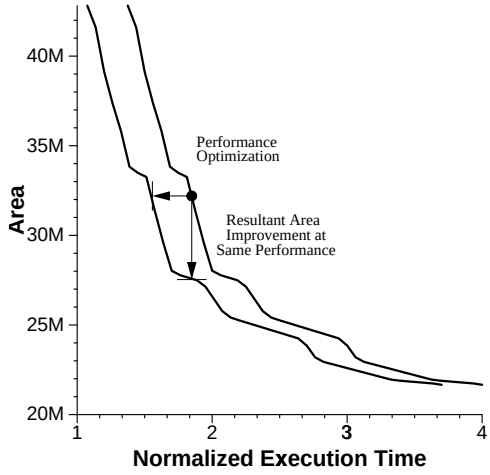

Figure 16: This is an example to show the relationship between performance optimizations and their effect on area. Performance optimizations can reduce execution time and at the same time can significantly reduce the amount of area needed to meet a given performance constraint.

been designed to optimize.

5.2.1 Optimization Tradeoffs

The goal of Sherpa is to use it to not only help guide the design of the architectural components already discussed in this paper, but to be able to quickly evaluate new architecture optimizations. This allows the designer to see the significance of their ideas to the overall processor design when trying to minimize area.

To show this, we investigated the application specific optimization by Sherwood and Calder [25] for automatically creating custom finite state machine (FSM) predictors for individual branches. This optimization augments the branch predictor with custom branch prediction entries. These entries have hard coded to them given branch addresses, and a custom finite state machine tailored to that branch. We include the custom FSM predictor into the Sherpa constraint system as another possible option for the branch predictor. Figure 15 shows the percent area reduction in relationship to performance when evaluating the use of custom finite state machines for branch prediction for `gs`.

The reason for the large reduction in area for the lower execution times comes from the increase in performance provided by the custom FSM branch predictor. Figure 16 graphically shows the reason for this by showing the relationship between performance optimizations and area. Optimizations that reduce execution time can at the same

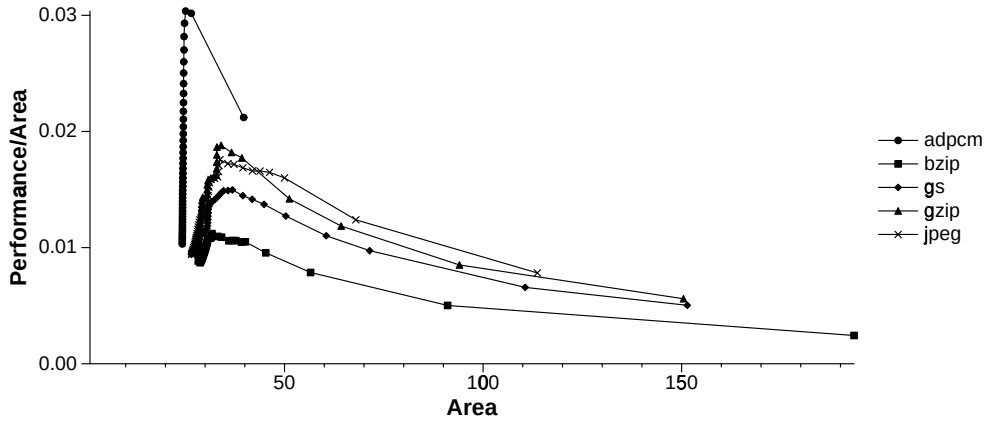

Figure 17: Performance per unit area of the different benchmarks shown as function of total chip area. While performance per unit area may not be useful to an embedded designer, seeking to meet some performance constraint, it is a result that could be easily used by chip multiprocessor designers to help maximize to total processing power of a given chip real-estate.

time significantly reduce the amount of area needed to meet a given performance constraint.

5.2.2 Performance/Area Versus Total Area Tradeoffs

Another sort of tradeoff analysis that can be provided by our system is computational efficiency analysis. Figure 17 shows a plot of performance per-unit area versus total area. In this plot we can see that the best performance efficiency for most of the programs we examined lies between 30 and 40 million square features, less than twice the size of the core functionality of the chip. While performance per unit area may not be useful to an embedded designer seeking to meet a given performance constraint, it is a result that would be very useful to chip multiprocessor designers in helping to maximize total processing power of a given chip real-estate. Application targeted chip multiprocessors, such as the Piranaha project [5], which was targeted at transaction processing, seeks to get higher performance by clustering together many simple processors with high computational efficiency onto a single chip. Our constraint system could be used to guide the design of the various subsystems on the chip to maximize the total performance.

6 Related Work

There is a great deal of related work in hardware/software co-design, application specific processor generators, and analytic performance estimation, but we have found none that address the problem we are attempting to solve. While a full listing of related work is not feasible in a paper of this length, we do point out some representative papers from each area and describe how our research fits into the broader context.

6.1 Configurable Cores

There are currently many configurable core designs, that can be customized or otherwise modified to application specific requirements. The XTensa processor [13] allows the designer to input specifications that they want in a processor, and the XTensa tools generate the needed components ready for integration into a system-on-chip. Another example of a configurable RISC core is the LX4380 [26] processor core. LX4380 supports adding up to 12 new instructions to the core as needed by a designer. The ARC processor core [4] is similar in design and intent to the LX4380. The tools available for these three configurable cores do not assist the designer in finding good design points for their application. We provide an automated system that will examine program behavior and suggest near-optimal design decisions, allowing the designer to make informed decisions early and experiment with different optimizations.

6.2 Design Exploration for Application Specific Processors

An approach being examined for designing customized processors is to automatically generate trial processors (with a machine description language [14, 19], a GUI [15], or a template as described above) for a specific application, and then examine their performance and feed this evaluation back into the automated processor generator.

The Lx [9, 10, 8] customizable VLIW architecture builds custom hardware for loop nests in applications. They have a clustered VLIW architecture that can be customized to a given application domain with a semi-automated optimization step. The system starts by generating a trial architecture, for which it generates compilers and other tools. Key parts of this application code are then compiled and measured, and are used to generate a new trial architecture, and the process is repeated.

The PICO project [1, 2] uses a fully automated approach to creating a customized processor. Their system starts by designing a set of Pareto optimal memory hierarchies, processors, and custom hardware accelerators. Then, different combinations of these points are tested by assembling the processor and simulating its performance in detail. The best combinations are noted and combinations that are similar to these are evaluated. They report that the system takes from 10 hours to several days.

The iterative approaches for Lx and PICO yield very good results, but take too long to be used interactively. We propose to use a high level modeling of performance first via Sherpa to allow the designer to very rapidly examine complex design tradeoffs in real time, and then feed this information into the iterative processor generation performed for Lx and PICO.

The Platune System [12, 11] has goals similar to ours, in that it too seeks to reduce the number of processor configurations that must be explored. The Platune System makes use of the fact that while some processor parameters are coupled, there are many others that are independent or are just one-way dependent. They then use this information entered by hand by the user to prune there design space, skipping over points that the user knows will not be Pareto optimal. The major difference between our two techniques is that after Platune prunes the design space, a brute force search of a parameters is still required which limits the overall number of parameter settings that can be searched. Sherpa, on the other hand, uses linear programming to model many discrete points that are near linear as a line segment which allows for the points to be searched analytically. We believe that a combination of the two techniques, using Platune to discover and group the dependent parameters and then exploring those parameters using Sherpa rather than exhaustive simulation can provide even better results than either technique by itself.

6.3 Performance Estimation

Another common field of study that is closely related to our work is analytical cache performance estimation. Agarwal et. al. [3] present an analytical cache model that can capture the behavior of cache systems based on the analysis of a trace. Both Whalley [30] and Quong [22] present models for capturing the expected instruction cache miss rate for a given application on different cache sub-systems either at compile time or through statistical analysis of a compressed trace. Jacob et. al. present work that uses a characterization of workload locality built from LRU stack distance to take an analytical approach to building memory hierarchies [16].

Many of these models could be incorporated into Sherpa to create the piece-wise linear functions for the different components we examined. Even so, we found using targeted simulation to quickly walk points in the design space to be efficient when generating the piece-wise linear functions.

7 Summary

Our approach to exploring this design space is to first divide it into separate regions mapped by largely independent parameters. These regions form individual sub-problems that can be efficiently and accurately modeled using data-driven analytical techniques. We formulate the results of these models into a single large constraint-based integer-linear program that can be efficiently solved using conventional mathematical techniques.

We showed that the results from solving of the constraint system lined up very closely with the actual performance numbers obtained through detailed simulation. We examined the tradeoffs that were made between the different components included in optimization and saw how they can play off each other in a complex manner. We further demonstrated how the Sherpa framework can be applied to rapidly evaluate potential optimizations and their impact on *both* the performance and area of the processor.

A goal of this research is to provide a more scientifically sound methodology for evaluating novel architectural techniques in the embedded space. The traditional method of proposing a new technique, and then examining the performance enhancement relative to a single baseline data-point is not very meaningful to overall system design in cost sensitive domains. An architect needs a way of visualizing the way new techniques trade-off against a range of potential design options. The methodology presented in this paper provides the ability to perform this range analysis. It places architectural changes in a global setting allowing the architect to gain a full picture of its usefulness.

To our knowledge we are the first to apply integer-linear programming to total-processor design space exploration. The significant research advance we contribute in this paper is the characterization of the processor design space into piece-wise linear models, the constraint formalism to combine them, our formation of piece-wise linear functions, and the validation of these techniques in this environment. The research in this paper lays the foundation for optimization of more complex architectures.

References

[1] S. Abraham, B. Rau, R. Schreiber, G. Snider, and M. Schlansker. Efficient design space exploration in pico. In *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 71–79, San Jose, California, November 2000.

[2] S. G. Abraham and S. A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *32nd International Symposium on Microarchitecture*, 1999.

[3] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[4] ARC. Whitepaper: Customizing a soft microprocessor core. http://www.arccores.com, 2001.

[5] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[6] M. Berkelaar. lp solve: a mixed integer linear program solver. ftp://ftp.es.ele.tue.nl/pub/lp_solve, September 1997.

[7] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[8] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable vliw embedded processing. In *27th Annual International Symposium on Computer Architecture*, pages 203–213, 2000.

[9] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

[10] Joseph A. Fisher. Customized instruction-sets for embedded processors. In *Proceedings of the Design Automation Conference, 1999*, pages 253–257, 1999.

[11] T. Givargis and F. Vahid. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computer Aided Design*, 21(11), November 2002.

[12] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *International Conference on Computer Aided Design*, November 2001.

[13] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March-April 2000.

[14] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *In Proceedings of the Design Automation Conference (DAC 99)*, pages 927–932, 1999.

[15] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. Effectiveness of the asip design system peas-iii in design of pipelined processors. In *In Proceedings of Asia and South Pacific Desing Automation Conference 2001 (ASP–DAC 2001)*, pages 649–654, 2001.

[16] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, 1996.

[17] E. Lawler and D. Wood. Branch and bound methods: A survey. *Operations Research*, 14(291):699–719, 1966.

[18] S. Leibson. Xscale (strongarm-2) muscles in. *Microprocessor Report*, September 2000.

[19] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language aidl. In *In Proceedings of Asia and South Pacific Desing Automation Conference 1997 (ASP–DAC 1997)*, pages 387–390, 1997.

[20] J. Mulder. An area model for on-chip memories and its applications. *IEEE Journal of Solid States Circuits*, 26(2):98–106, February 1991.

[21] M. Puig-Medina, G. Ezer, and P. Konas. Verification of configurable processor cores. In *Proceedings of the Design Automation Conference (DAC2000)*, pages 426–431, 2000.

[22] Russell W. Quong. Expected i-cache miss rates via the gap model. In *21st Annual International Symposium on Computer Architecture*, pages 372–383, 1994.

[23] G. Reinman and N. Jouppi. Cacti version 2.0. http://www.research.digital.com/wrl/people/jouppi/CACTI.html, June 1999.

[24] S. Santhanam. Strongarm 110: A 160mhz 32b 0.5w cmos arm processor. In *Proceedings of HotChips VIII*, pages 119–130, 1996.

[25] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *Annual International Symposium on Computer Architecture*, June 2001.

[26] C. Snyder. Synthesizable core makeover: Is lexra's seven-stage pipelined core the speed king? In *Microprocessor Report*, June 2001.

[27] C.D. Snyder. Fpga processors cores get serious. *Microprocessor Report*, 14(9), September 2000.

[28] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[29] Rabin A. Sugumar and Santosh G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 13(1):32–56, 1995.

[30] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems*, page 13, Newport, Rhode Island, USA, 1-5 1992.

[31] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, May 1996.

[32] Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: a fast flexible architecture for secure communication. In *28th Annual International Symposium on Computer Architecture*, pages 110–119, 2001.