Application-Specific Customization of Parameterized FPGA Soft-Core Processors

David Sheldon*, Rakesh Kumar†, Roman Lysecky‡, Frank Vahid*, Dean Tullsen†

*Department of Computer Science and Engineering University of California, Riverside **{dsheldon,vahid}@cs.ucr.edu**	†Department of Computer Science and Engineering University of California, San Diego **{rakumar,tullsen}@cs.ucsd.edu**	‡Department of Electrical and Computer Engineering University of Arizona **rlysecky@ece.arizona.edu**

ABSTRACT

Soft-core microprocessors mapped onto field-programmable gate arrays (FPGAs) represent an increasingly common embedded software implementation option. Modern FPGA soft-cores are parameterized to support application-specific customization, wherein pre-defined units, such as a multiplication unit or floating-point unit, may be included in the microprocessor architecture to speed up software execution at the expense of increased size. We introduce a methodology for fast application-specific customization of a parameterized FPGA soft core, using synthesis and execution to obtain size and performance data in order to create a tool that can be used across a variety of tool platforms and FPGA devices. As synthesizing a soft core takes tens of minutes, developing heuristics that execute in an acceptable time of an hour or two, yet find near-optimal results, is a challenge. We consider two approaches, one using a traditional CAD approach that does an initial characterization using synthesis to create an abstract problem model and then explores the solution space using a knapsack algorithm, and the other using a synthesis-in-the-loop exploration approach. We compare approaches for a variety of design constraints, on 11 EEMBC benchmarks, using an actual Xilinx soft-core processor, and for two different commercial Xilinx FPGA devices. Our results show that the approaches can generate a customized configuration exhibiting roughly 2x speedups over a base soft core, reaching within 4% of optimal in about 1.5 hours, including complete synthesis of the soft-core onto the FPGA, compared to over 11 hours for exhaustive search. Our results also show that including synthesis-in-the-loop, compared to a traditional CAD approach, improved speedups by an average of 20% when size constraints were tight. The approaches may also be applicable to soft-core processors targeted to ASICs in addition to FPGAs.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: – *Microprocessor/microcomputer applications, Real-time and embedded systems.*

General Terms

Performance, Design, Experimentation.

Keywords

Tuning, customization, FPGA, soft-core processors, parameterized platforms.

1. INTRODUCTION

Microprocessors on field-programmable gate array (FPGA) chips are becoming an increasingly popular software implementation platform, due to their coexistence on-chip with custom logic. Such coexistence can reduce parts costs and board sizes, and can improve system performance due to reduced communication times between processor and FPGA. A hard-core processor is laid out on the chip next to the FPGA's configurable logic fabric [2][4][18]. In contrast, a soft-core processor [3][17] is synthesized onto the FPGA's fabric, just like any other circuit. Compared to hard-core microprocessors on some FPGA devices, soft-core processors have the advantages of utilizing standard mass-produced and hence lower-cost FPGA parts and of enabling a custom number of microprocessors per FPGA (subject to size constraints) – over 100 soft-core processors can fit on modern high-end FPGAs. However, soft-core processors have the disadvantages of reduced processor performance, higher power consumption, and larger size.

While any microprocessor soft-core could conceivably be mapped to an FPGA, FPGA vendors have in the past few years introduced soft-core processors specifically targeted for FPGA implementation. Such FPGA soft-cores have instruction sets, arithmetic-logic units, register files, and other features specifically tailored to efficiently use FPGA resources, or perhaps more accurately, to avoid inefficient use of FPGA resources that may occur when synthesizing a general soft-core processor to an FPGA. The performance overhead of such soft-core processors on FPGAs compared to general soft-core processors on ASICs (application-specific integrated circuits) can thus be significantly less than the overheads when comparing FPGA versus ASIC implementations of general circuits.

A feature of FPGA soft-core processors is that of core configuration by the user (the application developer) through the setting of parameters. Configurable parameters may include instantiating a cache (and specifying its size), or instantiating a predefined datapath unit (like a multiplier or floating-point unit) and an accompanying instruction that uses the instantiated unit. Parameterized soft cores represent a different problem from that of developing custom datapath units and accompanying custom instructions, as done in application-specific instruction-set processors (ASIPs) like the ASIC-oriented ASIPs [15] or FPGA-oriented ASIPs [5], due to the "on/off" (or limited number of) values of the parameters. Yiannacouras [20] showed, using Altera FPGAs, that tuning a parameterized soft-core processor to an application could yield significant performance/size

improvements and reduced overall size versus a processor tuned to be best on average.

The contribution of this work is in developing effective approaches for automatically customizing a parameterized soft-core processor to an application. Presently, FPGA soft-core processor users must manually determine the best core configuration for a software application. Such manual configuration either results in unduly long exploration times due to evaluating too many configurations, or results in a sub-optimal configuration. We consider two approaches, a traditional CAD approach that maps to an abstract problem model and then solves the problem thoroughly while relying on estimations, and a synthesis-in-the-loop approach that uses actual synthesis and execution during exploration but searches only a fraction of the solution space. While our work's motivation lies in soft cores for FPGAs, our approaches may apply to ASICs also.

2. SOFT-CORE FRAMEWORK AND EXPLORATION METHOD

We developed our methodology using a Xilinx MicroBlaze FPGA soft-core processor [17], but the methodology would be applicable to other FPGA soft-core frameworks. The Xilinx MicroBlaze, referred to hereafter as MB, is a 32-bit soft-core processor designed for efficient implementation on Xilinx FPGAs. The MB is a single-issue in order execution processor. The MB can be configured to instantiate any combination of the following five components: multiplier, barrel shifter, divider, floating-point unit (FPU), and data cache. The first four components are each "on/off" type, either being instantiated or not instantiated, and only one instance of each component type is allowed (due to the MB being single-issue). The data cache, when instantiated, can be 2 Kbyte, 4 Kbyte, or 8 Kbyte, but we only consider 4 Kbytes in this paper for simplicity. Furthermore, the MB supports two cache types, an older basic cache, and a newer better performing "MCH" cache, although we only consider the latter. We thus consider $2^5=32$ possible MB configurations. When any of the first four components is instantiated, the MB includes a special instruction that uses that component (e.g., a multiply instruction), and the MB compiler generates code utilizing that special instruction. We refer to a *base MB* as an MB with none of the five extra components instantiated, and a *full MB* as an MB with all five components instantiated.

Instantiating a component increases an MB's size, but may improve an application's performance, depending on the application. We define the task of *customizing* a MicroBlaze for a particular software application as the task of instantiating a particular combination of components, known as a *configuration*, such that design goals, which may involve performance and/or size, are best met for an application running on the customized MB.

We measure performance as the time to execute an application once from beginning to end (typically an embedded benchmark application loops back to its beginning after the end). That time is the number of clock cycles multiplied by the clock period, referred to hereafter as the *application runtime*. We utilized Xilinx ISE and EDK tools to determine the clock period by synthesizing a configured MB onto a specific FPGA device. We measured the number of clock cycles by executing an application on an MB mapped to the FPGA device, with the application slightly modified to communicate with a clock-cycle counting circuit. The cycle counting circuit non-intrusively counts clocks cycles

$$LUT_{Equivalent} = LUT_{Regular} + LUT_{Equivalent(Mult)} + LUT_{Equivalent(BRAM)}$$

$$LUT_{Equivalent(Mult)} = \#Mult_{Used} / \#Mult_{full\,MB} * LUT_{full\,MB}$$

$$LUT_{Equivalent(BRAM)} = sizeBRAM_{Used} / sizeBRAM_{full\,MB} * LUT_{full\,MB}$$

Figure 1: Equations for calculating *Equivalent LUT* value of a configured MB.

while the application executes, and does not affect the application's performance.

A basic measure of a soft core's size on an FPGA is the number of utilized lookup tables ($LUTs$)[1]. However, a soft core may also utilize hard-core FPGA resources, such as hard-core multipliers or block RAMs. To be able to straightforwardly plot and compare sizes of different soft-core configurations, we assign an equivalent LUT value to hard-core resources. We did so by first measuring the regular LUTs, hard-core multipliers, and block RAM utilized in a full MB. We then combine the individual size metrics into a single size metric representing *equivalent LUTs*. Figure 1 presents the equations for calculating equivalent LUTs for a given MicroBlaze configuration. Assuming each type of resource (LUT, hard-core multiplier, or block RAM) is of equal importance, Figure 2 lists the equivalent LUT values for each hardcore unit. Then for a given configured MB, the equivalent LUTs, $LUT_{Equivalent}$, is the sum of the regular LUTs, $LUT_{Regular}$, used for logic to support datapath components, the equivalent LUTs for hard-core multipliers, $LUT_{Equivalent(Mult)}$, and the equivalent LUTs for block RAMs, $LUT_{Equivalent(BRAM)}$. The equivalent LUTs for the utilized multipliers is equal to ratio of multipliers used, $\#Mult_{Used}$, to multipliers in a full MB, $\#Mult_{full\,MB}$, multiplied by the number of regular LUTs in a full MB, $LUT_{full\,MB}$. Likewise, the equivalent LUT for the utilized block RAMs is equal to ratio of block RAM used, $sizeBRAM_{Used}$, to block RAM in a full MB, $sizeBRAM_{full\,MB}$, multiplied by the number of regular LUTs in a full MB, $LUT_{full\,MB}$. Of course, a user can weigh regular LUTs, multipliers, or block RAMs more heavily if that resource happens to be more valuable to the user. We recently noted that another research group working closely with Altera independently developed a similar equivalent LUT concept for similar size comparison purposes [20], thus lending confidence to the use of the equivalent LUT size metric during soft-core exploration. All LUT data in this paper represents equivalent LUTs. Interestingly, we recently discovered that our equivalent LUT concept correlates almost perfectly with Xilinx's own equivalent gate concept.

Note that the equivalent LUT concept is essentially a cost function that combines three terms by normalizing them and weighing

Component	Equiv LUT Count
LUT	1
MULT 18x18	569
BRAM	1328

Figure 2: Equivalent LUT values for hard-core units.

---

[1] We originally utilized configurable logic blocks (CLBs) as a measure of size, but MicroBlaze designers at Xilinx informed us that LUTs are a more accurate and useful measure.
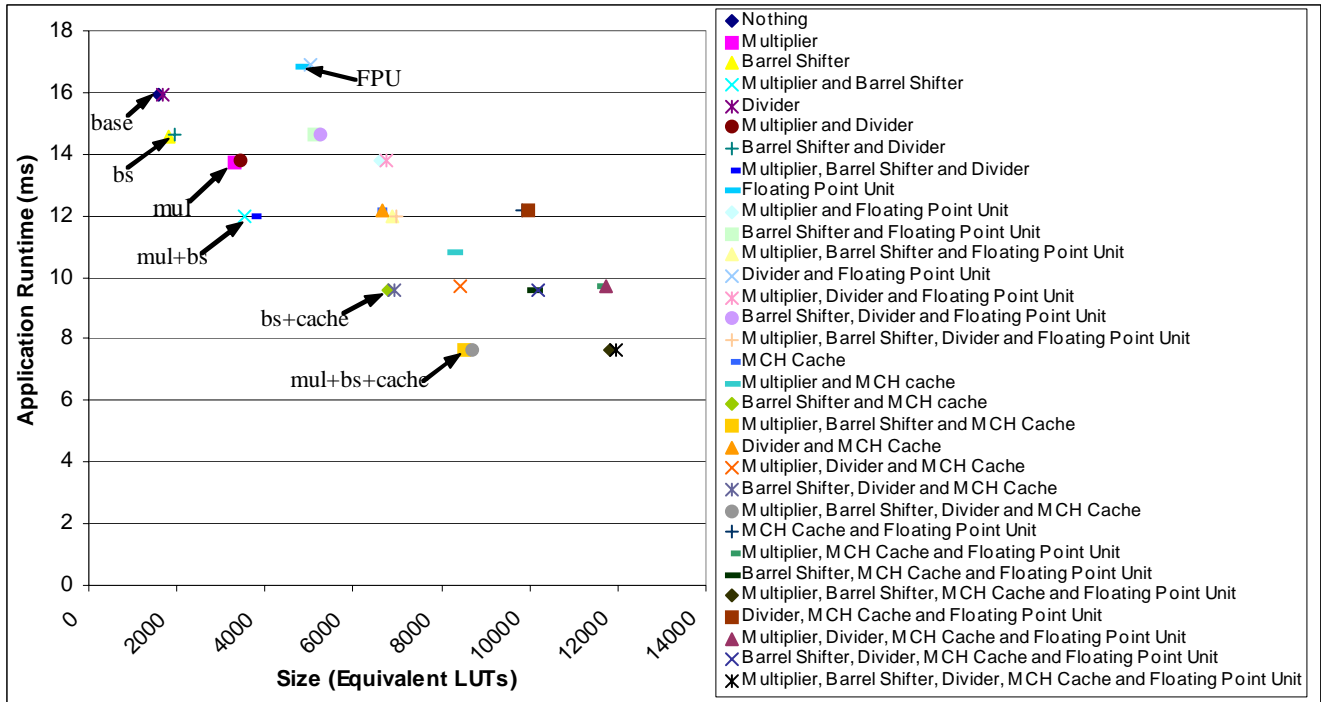
Figure 3: Size versus application runtime for all MicroBlaze configurations executing the *aifir* EEMBC benchmark, with all Pareto points labeled. An additional labeled point *(FPU)* is highlighted to show the performance overhead of instantiating an underutilized component, due to lengthening of the clock cycle.

them equally. Our approach is not strictly dependent on the above-described cost function; other functions could be used, including an approach where users specify the relative weights, or where different normalization methods are used.

In our experiments, we considered 11 benchmarks essentially selected at random from EEMBC [6], a benchmark suite intended for embedded systems. We report data for all of the randomly selected EEMBC benchmarks that we were able to compile and execute on the Xilinx MicroBlaze. In addition, we also considered an internally developed ray tracing application (*raytrace*) that is predominantly a floating-point application.

For each benchmark, we utilized scripts to run our search heuristics, where those scripts automatically performed FPGA synthesis and executed the application whenever necessary. The scripts execute on a computer connected to an FPGA development board (an ML310 board in our case). While the use of synthesis and execution may be viewed as a strength of our approach, it may also be viewed as a limitation, as there may be situations when a user wishes to explore but does not have a development board. A different approach involving pre-characterized cores, possibly combined with a soft-core simulation, might be necessary in that case. Alternatively, there may be a situation where an application cannot readily be run from scripts, such as when the application's execution requires human-generated input/output. In this case, our search approach could be supplemented with human interaction during the execution phases of exploration.

Figure 3 demonstrates the benefits of customizing an FPGA soft-core processor for one application. The figure presents the application runtimes for the EEMBC benchmark *aifir* running on

each of the 32 possible MB configurations. Considering only the Pareto-optimal configurations, the MB configurations have a 2X variation in application runtime and a 2X variation in LUTs, clearly demonstrating the benefits of configuring the MB to a particular application and its performance and size constraints.

Figure 4 presents the performance speedups of performance-optimal configured MB for all 12 benchmarks, as determined by exhaustively examining all possible configurations for each application. The optimal MB configuration on average has a 3.5x speedup compared to a base MB and a maximum speedup of 11.1x for the application *matmul*. However, obtaining that data by performing exhaustive exploration for this application required approximately 15 minutes per configuration (with 99% of that time spent on synthesis and with certain configurations requiring more than 15 minutes), resulting in over eleven hours of exploration tool runtime. Even for the relatively small number of
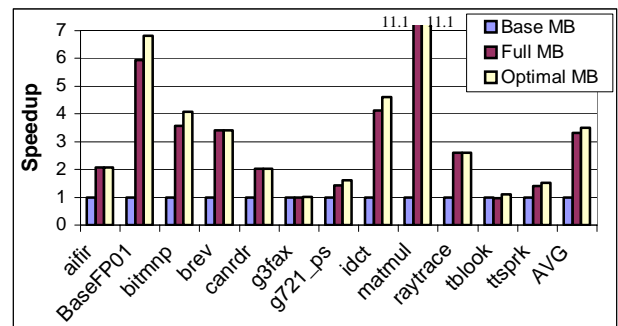

Figure 4: Speedups for base *(Base MB)*, full *(Full MB)*, and optimal *(Optimal MB)* MicroBlaze configurations.

263

Component	Cache	Floating Point	Divider	Multiplier
Barrel Shifter	5.2 %	1.0 %	0.0 %	10.4 %
Multiplier	6.7 %	1.9 %	26.0 %	
Divider	2.9 %	0.0 %		
Floating Point	5.1 %			

Figure 5: Average pairwise speedup-increment additive inaccuracies for all pairs of benchmarks.

configurable options we considered, exhaustively evaluating all possible configurations is already quite prohibitive, as a core user would need to re-evaluate all configurations anytime significant changes, and potentially even small changes, are made to the application, a common occurrence in a software design cycle. Furthermore, we expect that the number of configurable options will continue to increase for soft-core processors. As such, if the configurability is doubled from five options to ten options, the execution time for an exhaustive evaluation increases from approximately 11 hours to 11 days.

We sought to develop methods that would execute in approximately 1-2 hours – a tool runtime that we believe FPGA designers will find reasonable during the optimization step of design. Due to using synthesis during exploration, and because synthesis takes on the order of tens of minutes, the key feature of our developed heuristics must be that of executing only a few synthesis runs such that total customization time is on the order of 1-2 hours.

3. SOFT-CORE CUSTOMIZATION FOR APPLICATION RUNTIME

We consider the common goal of customizing an MB to minimize a particular application's runtime, with and without a size constraint. Fast tuning of configurable hardware platforms has been the subject of several recent research efforts. Most efforts assume that hundreds or thousands of configurations can be examined [1][7][9][10][11][13][14], but the 15 minute synthesis time in the FPGA soft-core problem means that only about 5-15 synthesis runs can be conducted.

3.1 Traditional CAD Approach: 0-1 Knapsack

We first considered developing a traditional CAD approach to tuning soft cores. The approach pre-characterizes the application and soft core, maps the problem to an abstract (and inexact) model, and then thoroughly solves the problem model.

We observed that the soft-core configuration problem could be approximately cast to a 0-1 knapsack problem, wherein one seeks to maximize the value of items placed in a knapsack having a weight constraint, with each item having a value and a weight. In

the fractional knapsack problem, one can include any fraction of items, while in the 0-1 knapsack problem, the only allowed fractions are 0 or 1, meaning the items are indivisible. We consider each optional MB component as an indivisible item. We assign a component's value to be the ratio of the speedup increment that occurs when instantiating that component compared to a base MB (e.g., a speedup of 1.4 has an increment of 0.4), over the size increment compared to a base MB. Note that the speedup increment for a component depends on the application, but the size increment is application independent. This cast is approximate, because speedup increments may not always be strictly additive when multiple components are instantiated. For example, component A may have an increment of 0.4 and B of 0.3, but A and B together may only yield an increment of 0.6, not 0.7. Likewise, size increments may not be strictly additive.

Figure 5 presents the inaccuracy of the additive assumption for all pairs of components. The additive assumption holds well (near-zero inaccuracy) for four pairs of components. Adding multiplier and barrel shifter speedup increments yields 10% inaccuracy, due to some shifts being achievable with a multiplier, and vice versa. Adding multiplier and divider speedup increments yields 26% inaccuracy.

A well-known optimal algorithm for solving the 0-1 knapsack problem first sorts items by their value/weight ratio, and then finds the optimal solutions using a dynamic programming algorithm [16]. To execute that algorithm, we must first compute the speedup increment (value) for each component. As that speedup is application dependent, we first execute six synthesis/executions, for the base MB, for the MB with a multiplier only, for the MB with a barrel shifter only, with an FPU only, with a divider only, and finally with only a cache. Figure 6 shows speedup increments, size increments, and their ratios, for the *aifir* EEMBC benchmark application.

The dynamic programming algorithm has what is known as a "pseudo-polynomial" runtime complexity of $O(n*W)$, where n is the number of items, and W is the knapsack weight constraint. This algorithm is known to be fast when W is a "small" integer, with a magnitude of perhaps $10,000 - 1,000,000$, and of course when n is also small. Fortunately, W is indeed a small integer in the case of our MB configuration problem (a full MB is only 12,000 equivalent LUTs) and n is of course small in our problem (5 instantiable units).

This approach applies six synthesis/execution runs when initially determining the component speedup and size increments, requiring about an hour, which dominates the approach's runtime. The inputs to the dynamic programming algorithm – n (number of soft core parameters) and W (number of available LUTs) – can each accommodate large increases before the 0-1 knapsack
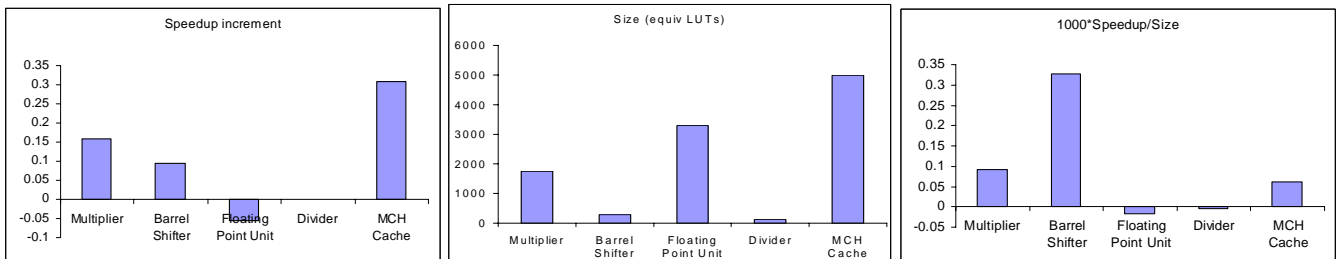

Figure 6: Speedup increment, size increment, and their ratio, for each MB component for the *aifir* benchmark.
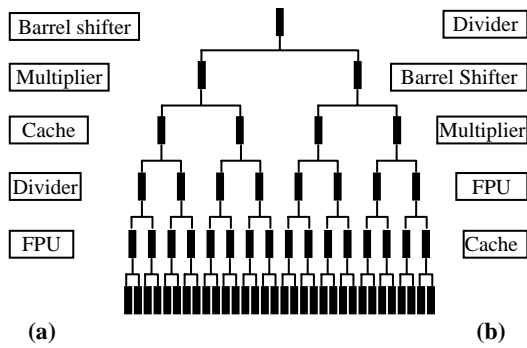
Figure 7: Impact-ordered tree approach: (a) application-specific impact-ordered tree for the *aifir* benchmark, (b) fixed impact-ordered tree. Note that *neither approach actually generates the entire tree* – both make a single descent to a leaf node.

algorithm runtimes approach a non-negligible time (versus synthesis) of tens of minutes. Even then, we have found that we can "quantize" the knapsacks weights by dividing all weights by 10 to yield a 10x algorithm speedup with almost no degradation in quality of results.

3.2 Synthesis-in-the-Loop Approach: Impact-Ordered Trees

Casting the soft-core configuration problem to 0-1 knapsack yields an approach with desired tool runtime and near-optimal results. However, the approach makes an assumption that speedup and size increments are additive, which is inaccurate for some pairs of components. As demonstrated in the later experiments section, those inaccuracies can result in sub-optimal solutions. We thus sought to also develop an approach that did not rely on the additive speedup increment assumption, but rather used synthesis/execution during exploration – synthesis-in-the-loop – while still executing just a few synthesis/execution runs.

We developed a greedy search method based on an approach proven effective in other parameterized architecture configuration research. The greedy method pre-determines the impact each parameter individually has on design metrics, and then searches the parameters in sequence, ordered from highest impact to lowest. For example, Zhang [21] used that method for customizing a highly configurable cache, where evaluating each configuration took many minutes due to lengthy simulations, and found near optimal results. We thus investigated such an impact-ordered approach.

The first phase of the approach determines the impact of each component. We can define impact simply as the speedup, but through experimentation, we found that a better definition takes

the ratio of speedup/size, just as in the knapsack problem. Thus, the first phase of the approach computes speedup increments, size increments, and their ratio, requiring six synthesis and execution runs, and resulting in the same data as in Figure 6. The second phase considers the components in order of their impact. For the current component, the approach instantiates the component, synthesizes and executes, and determines the application's runtime and size. If instantiating the component improves runtime and meets size constraints, the component is added; otherwise, it is not. The approach then moves on to the next component.

We refer to the above approach as an *application-specific impact-ordered tree approach*. Essentially, if we envision the entire solution space as a tree, as in Figure 7, the approach orders the levels of the tree, and then descends into only one sub-tree at each level, until reaching a single leaf node. The first phase orders the tree's levels, while the second phase makes a single descent. This approach requires six synthesis/executions for phase one, and five synthesis/executions for phase two, resulting in 11 total synthesis/executions.

We also investigated a variation of the above approach with the goal of reducing the number of synthesis/execution runs, by pre-determining average component impacts on a suite of typical benchmarks, rather than determining impacts on a per-application basis. The approach essentially moves phase one of the above approach from the tool user to the tool developer, thus cutting out six of the eleven synthesis/execution runs, leaving just five such runs. We refer to this approach as a *fixed-order impact-ordered tree approach*, because the impact ordering is fixed. Figure 8 shows the data averaged for all our benchmarks, with the speedup/size data resulting in the impact ordering shown in Figure 7(b).

3.3 No Size Constraint

Each of our algorithms assumes the problem we are solving is determining the best soft-core processor configuration given a limited size constraint. Some design scenarios impose no size constraint on the FPGA soft-core processor, instead seeking only the minimum application runtime. In the absence of a size constraint, one might assume minimal application runtime could be achieved by simply instantiating a full MB. However, this assumption is false, as was illustrated in Figure 4. Figure 4 presented the performance speedup for different MB configurations: a base MB, a full MB, and an MB configured for optimal application runtime (determine by exhaustive search) for the corresponding application compared to the base MB configuration. Notice for some applications that the full MB is actually slower than the optimal. The reason is because as more components are instantiated, the MB clock period may be lengthened, due in part to longer delays necessary for the increased wire routing within the larger MB. The point labeled
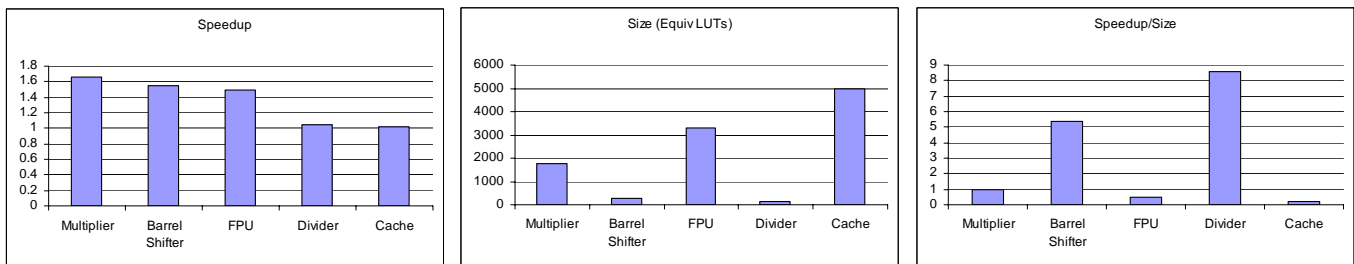

Figure 8: Speedup increment, size increment, and their ratio, for each MB component averaged across all 12 benchmarks.
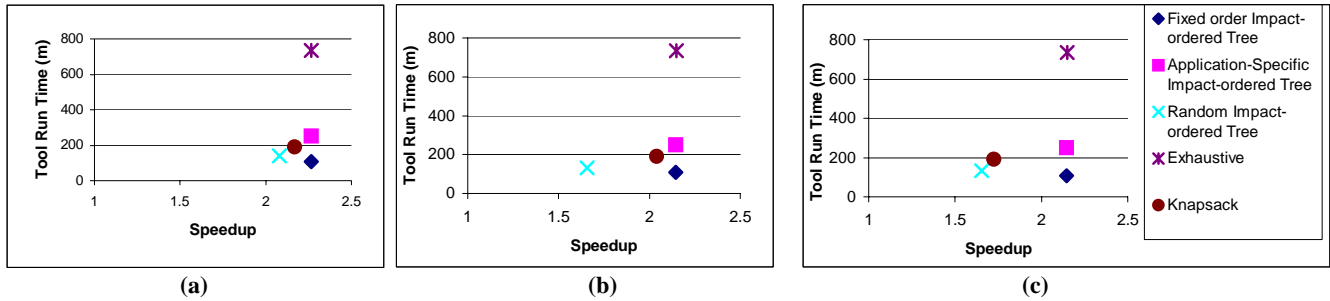
Figure 9: Average speedups obtained by the various exploration approaches, for: (a) no size constraint, (b) a fixed size constraint set at 80% of the size of a full MB, (c) a per-application-tailored size constraint of 80% of the size of the optimal MB for that application (as determined in (a)), all on a Virtex-II Pro device.

FPU in Figure 3 clearly illustrates the impact of longer delay caused by adding an underutilized FPU component.

To handle the no size constraint situations, in either the 0-1 knapsack approach or the impact-ordered tree approaches, we simply use a size constraint that is equal to (or larger) than the size of a full MB.

4. APPLICATION RUNTIME MINIMIZATION EXPERIMENTS

We implemented the knapsack, application-specific impact-ordered tree, and fixed-order impact-ordered tree approaches as scripts executing with Xilinx Platform Studio synthesis tools, coupled with a Xilinx Virtex-II Pro FPGA development board (ML310), for all 12 embedded benchmark applications. To compare the approaches with optimal results, we also implemented an exhaustive search approach that simply performed synthesis/execution for all 32 possible soft-core configurations.

Figure 9(a) presents the average speedups and tool runtimes for each approach for the scenario of unconstrained size. Exhaustive search requires over 700 minutes (11 hours) and finds average speedups of 2.3. The knapsack approach finds near-optimal solutions with a speedup of 2.2. Both impact-ordered tree approaches find the optimal solution. The fixed impact-ordered tree approach had the fastest runtime of 108 minutes. However, the knapsack approach should actually have roughly the same runtime, as both approaches synthesize about the same number of configurations. One particular configuration examined by the knapsack approach, namely a base MB with a barrel shifter alone, happened to have an unusually long synthesis time. Such anomalous synthesis runtimes are an artifact of the nature of FPGA physical design heuristics. In general, one should assume that the knapsack approach and the fixed impact-ordered tree approach will have equally fast tool runtimes.

One might wonder whether *any* ordering of the tree levels in the fixed impact-ordered tree approach would in fact yield the optimal configuration. We thus implemented another heuristic using a random ordering – barrel shifter, cache, FPU, divider, multiplier. Figure 9(a) shows that this random impact-ordered tree approach performs worse, though for the unconstrained size problem this approach is actually somewhat competitive.

Figure 9(b) presents the average speedups and tool runtimes for each approach for a fixed size constraint, chosen to be 80% of the size of a full MB. We also obtained data for a 50% constraint, with similar results (not shown). The plot again shows that the

impact-ordered tree approaches find optimal speedups (2.2), the knapsack approach finds near-optimal solutions (2.0), and the random impact-ordered tree approach is no longer competitive.

We sought to see how each approach would perform in a scenario where the size constraint was tight enough to prohibit use of the best performing MB for a given application. We thus created a unique size constraint for each application. Figure 9(c) presents the average speedups and tool runtimes for each approach with a tailored size constraint being 80% of the best performing MB for each particular application (as determined through exhaustive search with no size constraint, and choosing the smallest among equally performing configurations). We also obtained data for a 50% constraint, with similar results (not shown). While the fixed and application-specific impact-ordered tree approaches find the optimal, the knapsack heuristic performs very poorly for this size constraint. We found that the reason for the knapsack's poor results is due to the inaccuracy of the additive speedup increment assumption, which caused sub-optimal combinations of components to be selected.

To further evaluate the effectiveness of the approaches, we re-implemented the entire set of experiments for a Xilinx Spartan2 FPGA. Figure 10 presents the average speedups and tool runtimes for each approach for the case of unconstrained size. Again, the impact-ordered tree approaches are the best performing approaches, but the approaches chose configurations that were slightly below optimal on average. The application-specific approach found the optimal configuration in 11 of 12 cases, with a 20% worsening in performance for only one application. The fixed approach also resulted in a 20% worsening of performance for that same application, along with a 10% worsening for another application, but overall found the optimal configuration in 10 of 12 cases.

From this data, the fixed-order impact-ordered tree approach seems preferable. Of course, one must consider that our fixed-order was determined from the very same 12 benchmarks that we then used to compare the approaches. To examine this issue, we used six randomly selected benchmarks to define the fixed ordering, and then applied the approaches on the other six benchmarks only. The fixed impact-ordered tree approach again found the optimal for the constraint situations of Figure 9(a), (b), and (c), and even found the optimal for the situation of Figure 10. (Interestingly, the knapsack approach appeared markedly worse for that particular subset of six benchmarks). Of course, applying a particular fixed order on a radically different benchmark may yield worse results. Vendors might address that situation by having different fixed orderings for different application domains
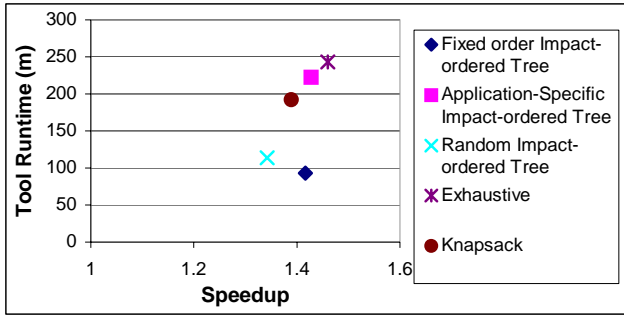
Figure 10: Average speedups for the approaches on a Spartan2 FPGA.

Figure 11: Pareto Optimal points for *aifir* benchmark running on the Virtex-II Pro FPGA

(e.g., control, signal processing, etc.), allowing the user to select a domain.

The application-specific impact-ordered tree approach is more robust in the presence of new benchmarks, but at the expense of about twice the tool runtime.

5. ADDITIONAL CONSIDERATIONS
5.1 Pareto Optimal Points Generation
Rather than minimizing application runtime (possibly with a size constraint), a designer may instead wish to obtain a set of possible design configurations that represent tradeoffs among application runtime and size. The configurations that represent meaningful tradeoffs – those for which no other configuration exists that is better or equal in both runtime and size – are known as Pareto points. For any reasonable design goal that combines performance and area, the Pareto points represent the only configurations that need to be considered. One approach to generating Pareto points is to exhaustively generate points for all possible configurations, and then remove all non-Pareto points. However, exhaustively examining all possible configurations may be too slow.

We instead utilize a heuristic method proposed by Givargis et al. [8] specifically for the purpose of finding Pareto points for parameterized system-on-a-chip platforms with configurable parameters in the cache, bus, and processor. That method heuristically prunes the search space by first exhaustively finding Pareto points for inter-dependent parameter subsets only, and then by composing the sets of Pareto points into a single set using an exact composition algorithm. The approach is heuristic because defining inter-dependent parameter subsets is inexact – parameters outside a particular subset may actually have some small dependencies with those in the subset. Givargis showed high accuracy, with search space pruning of over 99%.

We adopt this approach to our problem by defining inter-dependency as having speedup overlap beyond some threshold – in other words, as having a large speedup increment additive inaccuracy (>10%) as was presented in Figure 5. From that data, the barrel shifter and multiplier would be seen as inter-dependent, as well as the multiplier and divider. Thus, all three of those components form one inter-dependent subset. For the heuristic's first phase, we examine all component pairs for inter-dependency, resulting in the inter-dependent component subsets. In the second phase, we exhaustively evaluate all possible configurations of the inter-dependent subsets, for which we would examine all eight configurations of the three inter-dependent components. Note, however, that seven of those eight configurations were already examined in the first phase, and thus only one new configuration
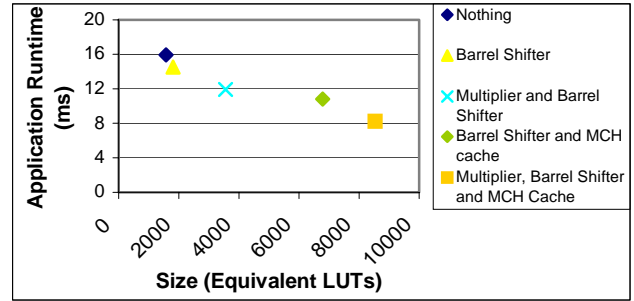
(the one with all three components instantiated) needs to be examined in the second phase for this inter-dependent subset. The remaining components form their own subsets, and we examine both configurations of each subset – again, both such configurations have already been evaluated in the first phase.

The complexity of the first phase is $O(n^2)$, where n is the number of components, but the complexity of the second phase is exponential. While the overall complexity is exponential, in practice the inter-dependent subset determination yields extensive pruning. Palesi et al. [12] further extended this exploration approach to provide faster execution by heuristically searching in the second phase.

Figure 11 shows the Pareto points generated by the heuristic for the *aifir* benchmark on the Virtex-II Pro FPGA. The heuristic finds all but one of the Pareto points highlight in Figure 3. A core user can choose a configuration from among the various Pareto-optimal configurations to meet system constraints. For *aifir*, the Pareto-optimal configurations range from a base MB, to a MB with multiplier, barrel shifter, and cache that has a 2x performance improvement but 4x size increase compared to the base MB.

5.2 Problem Variations
Our formulation considered only two-valued ("on/off") soft core parameters. Some parameters may have more than two possible values, such as a cache component that may be instantiated in one of several different sizes, or a multiplier that may be instantiated in one of several different versions trading off size and performance. We could extend the 0-1 knapsack to consider such multi-valued parameters by considering each version of a component as a separate component, and then using a disjunctive knapsack formulation [19] that prohibits specific items from appearing simultaneously in the knapsack (corresponding to prohibiting two versions of the same component, such as prohibiting two caches or two multipliers). We could extend the impact-order tree approaches by adding more than just two branches at the tree level corresponding to the multi-valued parameter, and either exploring all parameter values at the level, or heuristically exploring a few. Of course, multi-valued parameters may increase runtimes.

Our formulation considered five components. One can expect the number of soft-core parameters to increase beyond five. Figure 12 shows estimated tool runtimes for five to twelve two-valued parameters. While the approaches are significantly faster than exhaustive methods, the application-specific impact-ordered tree approach's runtime does increase to nearly 10 hours for twelve parameters. In contrast, the fixed-order impact-ordered tree scales
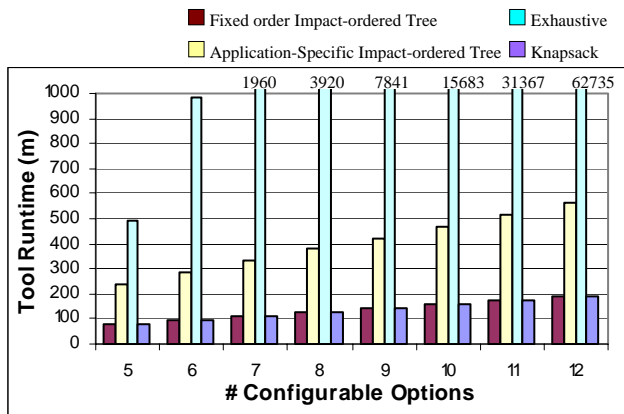
Figure 12: Estimated tool run times for increasing number of configurable soft-core processor options.

well, requiring just less than 3 hours for twelve parameters. Of course, the figure only shows runtime and not quality of results. We intend to investigate approaches for multi-valued parameters, and for more parameters, in future work.

6. CONCLUSIONS

We presented a methodology for automatically configuring FPGA soft-core processors. We considered two approaches: a traditional CAD approach, which pre-characterized applications, mapped to an abstract problem model, and used a 0-1 knapsack algorithm coupled with estimated size and performance values to optimally search the (inexact) solution space; and a synthesis-in-the-loop approach using impact-ordered tree search heuristics, which search only a fraction of the solution space but are guided by exact size and performance numbers. While the traditional CAD approach yielded good results, its reliance on estimation led to 20% sub-optimal results when we imposed tight size constraints. In contrast, the synthesis-in-the-loop approach yielded optimal or near-optimal speedups in all considered situations, while having competitive runtimes to the knapsack approach. The fixed-order impact-ordered tree synthesis-in-the-loop heuristic yielded near-optimal results in about 1.5 hours per application, but did poorly on a few examples. The application-specific impact-ordered tree approach demonstrated more robustness by yielding optimal or near-optimal results for all examples, but with runtimes of about 200 minutes. The fixed-order impact-ordered tree is acceptable due to good results with runtimes in our target of 1-2 hours, but the application-specific impact-order tree approach would be preferred if more runtime is available. Both heuristics should therefore be made available to a designer. Future work includes investigating soft cores with more parameters, and with more than two values per parameter.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] Abraham, S. G., B. R. Rau. Efficient design space exploration in PICO. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2000.

[2] Altera Corp. Excalibur Embedded Processor. http://www.altera.com/ products/devices/excalibur/exc-index.html, 2005.

[3] Altera Corp. Nios II Processors. http://www.altera.com/products/ ip/processors/nios2/ni2-index.html, 2005.

[4] Atmel Corp. FPSLIC (AVR with FPGA). http://www.atmel.com/products/FPSLIC/, 2005.

[5] Cong, J., Y. Fan, G. Han, Z. Zhang. Application-Specific Instruction Generation for Configurable Processor Architectures, FPGA 2004.

[6] EEMBC. http://www.eembc.org/, 2005.

[7] Givargis, T., F. Vahid. Platune: A Tuning Framework for System-on-a-Chip Platforms. IEEE Transactions on Computer Aided Design, Vol. 21, No. 11, Nov. 2002, pp. 1317-1327.

[8] Givargis, T., F. Vahid, J. Henkel. System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip, Intl. Conf. on Computer-Aided Design (ICCAD), 2001.

[9] Mishra, P., N. Dutt, and A. Nicolau. Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures. International Symposium on System Synthesis, 2001.

[10] Mohanty, S., Prasanna, V. K., Neema, S., and Davis, J. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. Joint Conference on Languages, Compilers and Tools For Embedded Systems, 2002

[11] Palermo, G., C. Silvano, S. Valsecchi, V. Zaccaria. A System-Level Methodology for Fast Multi-Objective Design Space Exploration. Great Lakes Symposium on VLSI (GLVLSI), 2003.

[12] Palesi, M., T. Givargis. Multi-Objective Design Space Exploration Using Genetic Algorithms. International Workshop on Hardware/Software Codesign (CODES), 2002.

[13] Sherwood, T., Oskin, M., and Calder, B. Balancing design options with Sherpa. 2004. International Conference on Compilers, Architecture, and Synthesis For Embedded Systems (CASES), 2004.

[14] Szymanek, R. F. Catthoor, and K. Kuchcinski. Time-Energy Design Space Exploration for Multi-Layer Memory Architectures. Design, Automation and Test in Europe (DATE), 2004.

[15] Tensilica, Inc. The XPRES Compiler: Triple-Threat Solution to Code Performance Challenges. http://www.tensilica.com/ pdf/XPRES-Triple-Threat_Solution.pdf, 2005.

[16] Toth, P. Dynamic Programming Algorithms for the Zero-One Knapsack Problem. Computing 25, pp. 29-45, 1980.

[17] Xilinx, Inc. MicroBlaze Soft Processor Core. http://www.xilinx.com/ xlnx/xebiz/designResources/ip_product_details.jsp?key=micro_blaze , 2005.

[18] Xilinx, Inc. Virtex-4 Platform FPGA. http://www.xilinx.com/ products/silicon_solutions/fpgas/virtex/virtex4/index.htm, 2005.

[19] Yamada, T., S. Kataoka and K. Watanabe. Heuristic and Exact Algorithms for the Disjunctively Constrained Knapsack Problem. Information Processing Society of Japan Journal, Vol. 43, No. 9 (2002), 2864-2870.

[20] Yiannacouras, P., J. G. Steffan, and J. Rose. Application-Specific Customization of Soft Processor Microarchitecture. FPGA 2006.

[21] Zhang, C., F. Vahid and R. Lysecky. A Self-Tuning Cache Architecture for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Vol. 3, No. 2, May 2004