Conjoining Soft-Core FPGA Processors

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

*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 programmable processors on field-programmable gate arrays (FPGAs) can be custom synthesized to instantiate only those hardware units, such as multipliers and floating-point units, that an application requires to meet performance demands, thus minimizing soft-core size on the FPGA. Conjoining processors, meaning to share hardware units among two or more processors, can further reduce soft-core size, leaving more resources for other circuits such as custom coprocessors. Using Xilinx MicroBlaze coprocessors and standard embedded system benchmarks, we show that conjoining two processors can provide 16% processor size reductions on average, with less than 1% cycle count overhead. We introduce an efficient dynamic-programming-based exploration method to find the best custom instantiation of hardware units, considering both standalone and conjoined options, for soft-core processors.

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

FPGAs, soft-core processors, conjoined processors, tuning, customization, parameterized platforms.

1. INTRODUCTION

Soft-core processors on field-programmable gate arrays (FPGAs) are an increasingly popular software implementation option in embedded computing systems. A soft-core FPGA processor is a synthesizable processor mapped onto the FPGA fabric, in contrast to a hard-core processor that is laid out next to the FPGA fabric. FPGA vendors tailor synthesizable processors, such as the Xilinx MicroBlaze or the Altera Nios, for FPGA implementation, resulting in processors having less size and performance overhead than a general synthesizable processor mapped to an FPGA [2][12]. Soft-core, as well as hard-core, processors on FPGAs enable reductions in system device counts by co-existing with custom processing circuits and glue logic on a single device. Soft-core processors possess an additional advantage of being realizable on general-purpose FPGA devices, with those devices typically being lower cost than devices with hard cores due to mass production and hence economy of scale. Furthermore, soft-core processors enable custom numbers of processors on a device, and custom interconnection structures among those processors – increasingly important features as multiprocessing systems grow in importance and diversity.

Soft-core FPGA processors come with optional hardware units that may be instantiated, such as a multiplier, divider, barrel shifter, or floating-point unit. FPGA tools generate accompanying instructions, like a multiply instruction, to utilize an instantiated unit, and soft-core compilers then utilize those instructions rather than software library routines. Because a soft-core user may map a single software application onto a soft-core, the user typically instantiates minimal hardware units to meet desired performance targets while minimizing the soft-core's circuit size, a task known as *customizing* the soft-core. Such soft-core customization can reduce soft-core size by a factor of three compared to a core with all units instantiated. That reduction not only frees FPGA resources for use by other circuits co-existing on the FPGA, but also can enable use of smaller and hence lower-cost FPGA devices. Such reduction is magnified by the increasingly common situation of users mapping several or even dozens of soft-cores onto a single FPGA device [5][6][7][8][9], making soft-core customization even more critical to best utilize available FPGA resources. Furthermore, our analyses have shown that reducing hardware units, in addition to reducing circuit size, can even improve performance, due to shorter critical paths in the soft-core's circuit and hence faster processor clock frequency. Because FPGAs typically support numerous clock frequencies within a single device, each processor on a device could conceivably be clocked at its fastest frequency.

An important problem is to find, for a given application, the instantiation of possible hardware units that minimizes size while meeting performance constraints. Given the large number of possible configurations of hardware units, and the interactions among units, the problem is quite challenging.

We can distinguish between two types of customizable processors. A *customizable-instruction processor* allows definition of custom-built hardware units and accompanying custom instructions, and is often referred to as an application-specific instruction-set processor (ASIP). Examples include [1][3][4][10]0; Cong has investigated ASIPs specifically targeted to FPGAs [3]. A *parameterized processor* has specific pre-determined parameters that can be set to particular values to create a custom processor instance. One type of parameter corresponds to instantiating a pre-determined hardware unit and

ICCAD'06, November 5-9, 2006, San Jose, CA
Copyright 2006 ACM 1-59593-389-1/06/0011...$5.00

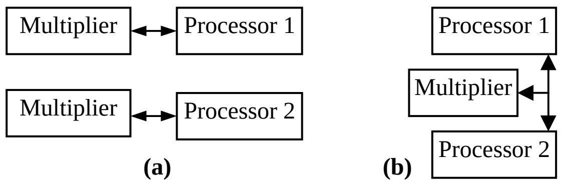

Figure 1: Conjoined processor example: (a) two separate processors each with a multiplier, (b) two conjoined processors sharing a single multiplier.

possibly an accompanying instruction, such as a multiplier unit and a multiply instruction, or a floating-point unit (FPU) and FP instructions. There may even be multiple versions of a unit, such as several multipliers that tradeoff speedup and size. Other parameters may relate to the size of the register file, the number of pipeline stages, the activation of data forwarding, the inclusion and configuration of cache, etc. Yiannacouras et al [14] showed the performance and size benefits of application-specific tuning of parameterized FPGA soft-core processors. The two types of customizable processors are not distinct; for example, Altera's Nios soft-core processor supports both types of customization [2]. In this paper, we specifically consider parameterized processors consisting of pre-defined hardware units.

Conjoining processors means to share hardware units between two (or more) processors. For example, two processors might share a single multiplier as shown in Figure 1(b), thus reducing total size of the two-processor system compared to each processor having its own multiplier as in Figure 1(a). Conjoining may result in performance overhead due to extra multiplexing and longer routing caused by conjoining. Performance overhead may also be caused by extra clock cycles due to contention for the shared unit, because if one processor requires use of the shared unit but another processor is already using the unit, the first processor may stall until the unit becomes available. Kumar [8] introduced conjoined processors and showed their benefits in multi-core desktop processor architectures. To the best of our knowledge, no soft-core FPGA vendor today supports conjoined processors. Thus, the data presented in this paper presents a case for future support of conjoinment. Conjoining processors may become an increasingly important consideration as soft-core processors continue to gain additional optional hardware units, such as multiply-accumulate units, vector processing units, and signal/image processing units like sum-of-absolute difference units. Furthermore, even while Moore's Law eases size constraints, smaller FPGA devices tend to be lower cost and lower power, making system size minimization an important goal for multiprocessor systems implemented on FPGAs.

In this paper, we provide results of an analysis showing that conjoining soft-core FPGA processors would yield very little clock cycle count overhead, while achieving significant size reductions, for a commercial soft-core processor executing standard embedded system benchmarks. We also develop an effective exploration method to automatically customize a two-processor system of parameterized processors, considering non-conjoined as well as conjoined options for every unit.

	Regular LUTs	Hard-core mult 18x18	"Size" (Equivalent LUTs)
Barrel Shifter	228	0	228
Divider	122	0	122
Multiplier	41	3	1331
Floating Point Unit	1018	4	2738
Base MicroBlaze	1570	0	1570
Full MicroBlaze	3010	7	5989

Figure 2: FPGA resource requirements for each instantiatable unit, and for the base and full MicroBlazes.

2. CONJOINED PROCESSOR ARCHITECTURE

Our experimental framework utilizes Xilinx MicroBlaze soft-core processors mapped to a Virtex II device on an ML310 board. The MicroBlaze is a parameterized soft-core processor, coming with the following hardware units that can be optionally instantiated using Xilinx's Embedded Development Kit toolset: a multiplier, a divider, a floating-point unit, and a barrel shifter. Upon instantiating one (or more) units, the toolset generates an accompanying instruction (or instructions), which the MicroBlaze compiler may then use when generating code for an application. Furthermore, the MicroBlaze also has instruction and data caches that can be instantiated, which have possible sizes ranging from 0 to 64 Kbytes. We will use the term *base MicroBlaze* to refer to a MicroBlaze with none of these optional hardware units instantiated, and the term *configured MicroBlaze* to refer to a MicroBlaze having a particular instantiation of the optional hardware units. A *full MicroBlaze* has all units instantiated. The toolset synthesizes a circuit for a configured MicroBlaze, with that circuit utilizing hard-core items on the FPGA when possible, such as hard-core multipliers (for the multiplier or floating-point units) or block RAMs (for cache). For such units, the toolset also synthesizes control logic circuits onto the FPGA fabric alongside e base MicroBlaze processor circuit, and some units, like the barrel shifter, consist entirely of such logic circuits on the fabric.

Figure 2 provides size data for each of the MicroBlaze's instantiatable hardware units and for base and full MicroBlazes. In discussing sizes, we need a straightforward way to describe the relative sizes of two configured MicroBlazes. Describing relative sizes is non-trivial because a MicroBlaze uses two types of hardware resources: lookup tables (LUTs), and hard-core multipliers. (We presently do not consider cache, so we do not consider an FPGA's block RAM hardware resources.) We thus define the concept of *Equivalent LUTs* for the multipliers for the purpose of describing relative sizes. A full MicroBlaze uses 3010 regular LUTs and 7 hard-core multipliers. Assuming regular LUTs and hard-core multipliers to be equally important resources, then the 7 hard-core multipliers have an equivalent LUT value of 3010, and one hard-core multiplier has an equivalent LUT value of 3010/7 = 430. An instantiatable unit or configured MicroBlaze thus has an equivalent LUT value equal to the number of regular LUTs used, *plus* the number of equivalent LUTs contributed by the hard-core multipliers used. For example, the floating point unit uses 1018 regular LUTs, and 4 hard-core multipliers worth

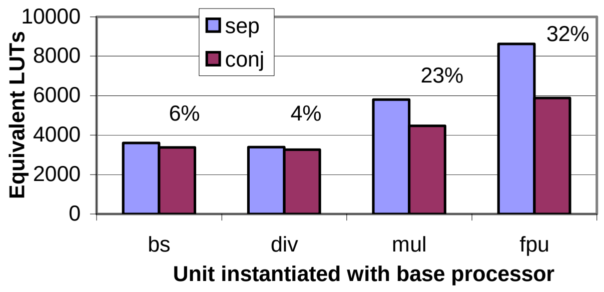

Figure 3: Sizes of two-processor systems with two separate units (sep), or one conjoined unit (conj). % size savings from conjoining are shown, showing significant savings.

430 equivalent LUTs each, for a total of 1018+4*430 = 2738 equivalent LUTs. A similar equivalent LUT concept was developed independently by researchers using Altera devices [14], lending confidence to the utility of the concept. We also recently found our equivalent LUT concept to correlate almost perfectly with Xilinx's own equivalent gate concept.

We use LUTs, and not configurable logic blocks (CLBs), as a measure of size, based on direct communications with the MicroBlaze design time indicating that LUTs are a more meaningful size measure than CLBs.

Using equivalent LUTs, Figure 3 shows the size savings achievable by conjoining just one unit for two processors. Conjoining two or more units would result in further size savings. Conjoining all units would yield 65% size savings. .

The next section addresses the key question of how much clock cycle count performance penalty is imposed by such conjoining.

3. CONJOINED PROCESSOR PERFORMANCE OVERHEAD

This section describes experiments to determine the clock cycle count performance overhead that would occur for standard benchmarks when conjoining two FPGA soft-core processors.

3.1 Simulation Framework

The MicroBlaze toolset includes a MicroBlaze simulator, which can generate instruction traces for any configured MicroBlaze. The MicroBlaze toolset does not support synthesis of conjoined processors (nor at this time does any soft-core processor toolset of any company that we are aware of), and hence that simulator does not simulate conjoined processors. We therefore developed a trace simulator that takes as input those instruction traces and configuration data including conjoinment information, and that outputs stall and cycle data. The trace simulator presently considers a two-processor system, and assumes that both processors operate at the same clock frequency. The trace simulator takes as input a list of conjoined hardware units. Each conjoined unit may have an access penalty associated with the unit, expressed in number of cycles. We pessimistically assume that every conjoined unit has a one cycle access penalty even when the unit is available, to account for checking of a busy flag. Schemes can be introduced to reduce the penalty well below one cycle on average [8].

Configuration		Cycle count
Barrel Shifter		2
Divider		34
Multiplier		3
FPU	Add, Sub, Mul	6
	Div	30

Figure 4: Cycle counts for each instantiatable unit when conjoined, including a one-cycle access penalty due to conjoinment.

A *collision* occurs when two processors need to use a conjoined unit in the same cycle. There are two types of collisions that can occur. The first is when processor A is already using a unit and processor B needs to use that unit. This type of collision can only occur for multi-cycle units. All of the components in the MicroBlaze are multi-cycle, as shown in Figure 4. For such a collision, our simulator assumes that processor currently using the unit continues to use the component until finished. The second type of collision occurs when both processors want to start using a unit on the same cycle. In this case, we use a simple arbiter that uses a round-robin (alternating in the case of two processors) policy. In either type of collision, we pessimistically assume that the processor waiting for a unit will completely stall.

Conjoining processors could potentially decrease the clock frequency of one or both processors, if the hardware required to share a conjoined unit lengthens the critical path. This decreased clock frequency could be minimized by placing the processors and the conjoined unit in such a way that the shared components are sitting between the two processors. As FPGAs do not presently support conjoined processors, we are presently unable to determine whether conjoining will actually impact frequency, and if so, to what extent. We plan to investigate this subject in the future through collaborations with an FPGA company.

3.2 Speedups for Instantiatable Units

We considered 10 applications from the EMBCC and Mediabench benchmark suites (*aifir*, *BaseFP01*, *brev*, *bitmnp*, *canrdr*, *g3fax*, *g721_ps*, *matmul*, *ttsprk*, *ttblook*) and one additional benchmark *raytrace*, for a total of 11 benchmarks. These benchmarks were chosen to show how the impact of the units varies over a wide range of benchmarks.

Each benchmark has a "beginning" and an "end," enclosed in a main loop whose iteration count can be varied. The beginning to end behavior may itself contain loops, but such behavior does not contain the infinite loop that often surrounds an embedded application. For each application, we first determined the number of cycles to execute the application from beginning to end (i.e., one main iteration) on a base processor. For each application, we also determined the number of cycles to execute one main iteration on a processor consisting of the base processor plus exactly one optional hardware unit, doing so for each possible hardware unit. Figure 5 shows the cycle data for one application, *aifir*, with that data consisting of the cycles on the base processor, and on all versions of the processor extended with one hardware unit. That figure shows that instantiating either a barrel shifter, or a multiplier, reduces the number of cycles needed to execute the application, and hence yields the speedups shown.

Configuration	Cycles	Speedup vs. base
Base MicroBlaze	2,134,921	1
Base + Barrel Shifter	1,833,752	1.16
Base + Divider	2,134,920	1
Base + Multiplier	1,849,715	1.15
Base + FPU	2,134,921	1

Figure 5: Cycle counts for one main iteration of the *aifir* benchmark, for a base MicroBlaze, and for a base MicroBlaze plus one unit. The barrel shifter and multiplier each provide speedups over the base for this benchmark.

Figure 6 shows the speedup data we obtained by each hardware unit for every application, in addition to *aifir*. That figure shows substantial speedups, as high as 6.5 in some cases. The next question is therefore how much of that speedup would be lost if the unit yielding the speedup were conjoined with another processor whose application also needs that unit for speedups.

3.3 Speedup Reductions due to Conjoining Units

We considered all possible pairs of applications running on a two-processor system, with one application per processor. Because applications have different runtimes, we increased the number of main loop iterations of the shorter application so that its runtime was longer than the longer application, and then we ran for the length of the originally longer application. For each application pair, we considered each optional hardware unit H. Considering the speedups shown in Figure 6, three possible situations exist among a pair of applications and the hardware unit H.

One situation is that neither application derives a speedup benefit from instantiating H. In this situation, neither processor would instantiate H, so conjoinment is not possible. A second situation is that only one application derives a speedup from instantiating H. In this case, only that processor might instantiate H, so conjoinment need not be considered.

The third situation is when both processors derive speedups from instantiating H. In this situation, conjoinment is an option to consider among five options: the processors share one H (conjoined), the processors each instantiate their own H, the first processor instantiates H but the second doesn't, the second processor instantiates H but the first doesn't, or neither processor instantiates H. This third situation, which we refer to as a *conjoinable situation*, is the only one for which we collected conjoinment data, because providing conjoinment data for the other two situations would have shown no speedup reductions and would have thus resulted in misleadingly low average performance overheads, i.e., in an exaggeration of the benefits of conjoinment.

For every conjoinable situation encountered when considering all pairs of applications and every hardware unit, we determined the cycles each application required from beginning to end, but this time assuming that the hardware unit was shared among the two processors running those two applications. Recall that we pessimistically assume *every* access to a shared unit, even in the absence of a collision, has a one-cycle access penalty incorporated in a unit's cycle latency in Figure 4. Because a shared unit could be busy when an application required that unit, the number of cycles for a particular application could increase over the cycles shown in Figure 4, i.e., sharing a unit may result in processor stalls. For example, Figure 7 shows the stalls computed by our trace simulator when sharing a barrel shifter between two particular benchmarks, chosen for the figure due to their resulting in one of the most stalls of any pair (note that the stalls are not very frequent even for that pair). Note that the one-cycle access penalty for a shared unit is also shown as a stall cycle.

We computed the speedups (over a base processor) for each application for every conjoinable situation considering each pairing with another application and each hardware unit, and compared those speedups with the earlier-computed speedups of Figure 6 for each hardware unit without any conjoining. We define the *performance overhead* of conjoinment as the ratio of the unconjoined speedup minus the conjoined speedup, divided by the unconjoined speedup, times 100%. Thus, if conjoining yields

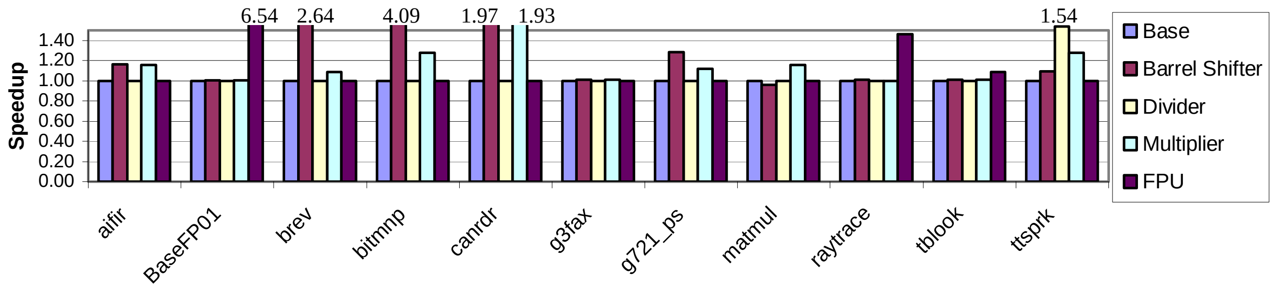

Figure 6: Speedups for each instantiatable unit, for each benchmark. Instantiating particular units can have significant speedup impacts.

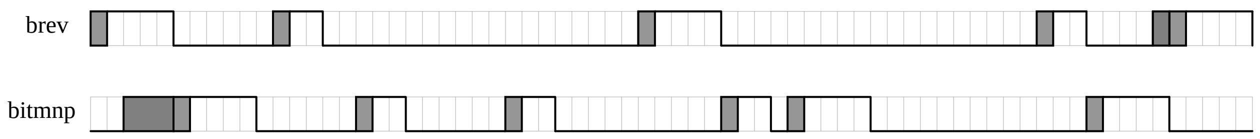

Figure 7: Stalls (shown as filled regions) determined by the simulator using a shared barrel shifter, for the brev, bitmnp pairing of benchmarks. This example involves one of the highest amounts of interference of all the examples considered.

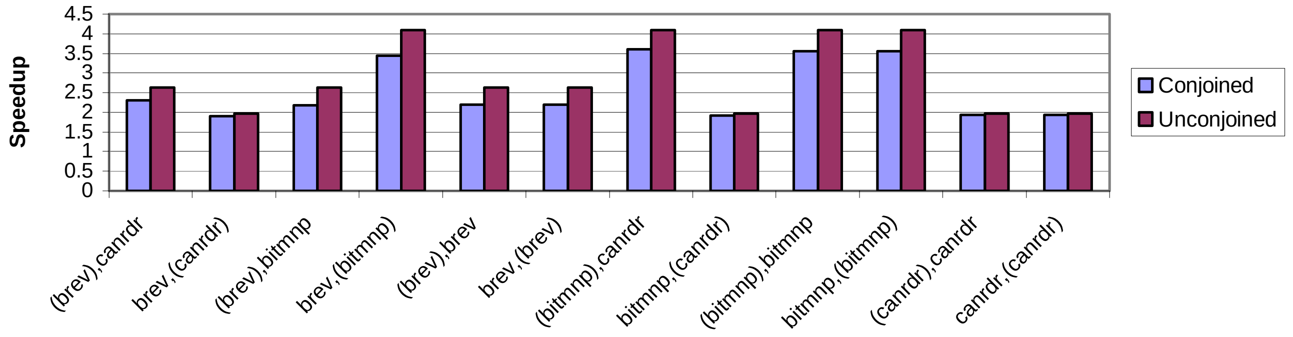

Figure 8: Application speedups for six pairings of applications, for conjoined and unconjoined barrel shifter cases. Only applications that benefit from the barrel shifter (i.e., for which an unconjoined barrel shifter provides a speedup of 1.3 or more) are shown. The benchmark in parentheses is the one from that pair whose speedup is shown in the bar above. The figure shows that conjoinment only has a small impact on the speedup provided by the barrel shifter.

a speedup of 1.7 whereas unconjoined execution yielded a speedup of 2.0, the overhead would be 100%*(2.0-1.7)/2.0 = 15%. We again point out that this performance overhead is with respect to cycle count only, and does not presently consider potential lengthening of the clock cycle caused by conjoinment.

Figure 13 shows the conjoinment performance overheads for all pairs of applications, for each hardware unit, only showing data for conjoinable situations (as defined earlier). The data shows that conjoinment results in very small performance overheads, usually 1% or less, occasionally about 5%, and only in a couple cases around 16% (which happened to involve a barrel shifter). In other words, most of the benefit of instantiating a unit is preserved even when the unit is conjoined.

Figure 8 shows the data specifically for a barrel shifter unit, and only for six “significant” pairs for which a barrel shift yielded speedups of 1.3 or more. Other pairs are omitted as they do not use the barrel shifter much and thus do not provide interesting data points. The figure again shows that conjoinment has only modest performance overhead. Plots for other units are similar, actually better – we showed the barrel shifter since it exhibited the most performance overhead compared to all other units.

Figure 9 shows the barrel shifter utilization for the same six pairs of benchmarks, showing that unit utilization is 40% on average, and over 50% in some cases. Yet even with such relatively high utilization, performance overheads were relatively small. The early example in Figure 7 for one of the highest overhead situations (*brev/bitmnp* sharing a barrel shifter) – even with 53% utilization (37 of 70 cycles), stalls are infrequent. Similar patterns occur for other example application pairs; in fact, other pairs exhibit even fewer stalls.

4. EXPLORATION METHOD FOR THE TWO-PROCESSOR UNIT-INSTANTIATION WITH CONJOINING PROBLEM

The previous section showed that conjoining soft-core processors could potentially yield significant size savings with small or no performance overhead in most cases. In this section, we introduce an automated exploration method for determining the best instantiation of units for two processors, with conjoining considered.

4.1 Problem Definition

Given a pair of applications running on the two processors, and a total size constraint, the *two-processor unit-instantiation with conjoining problem* is to find the instantiation of units that gives the greatest average speedup while not exceeding the size constraint.

We consider all four earlier-mentioned instantiatable unit types: barrel shifter, divider, multiplier, and FPU. Each unit type has five possible instantiations for two processors A and B: no instantiation, instantiated for A, instantiated for B, instantiated for A and instantiated for B (i.e., instantiated twice), and instantiated for both A and B (i.e., instantiated once but shared – conjoined). The complete solution space is thus 5*5*5*5, or 625 possible instantiations. Each unit has a size as shown in Figure 2. Each processor executes one application repeatedly. Performance speedup for a given instantiation of units is computed as described

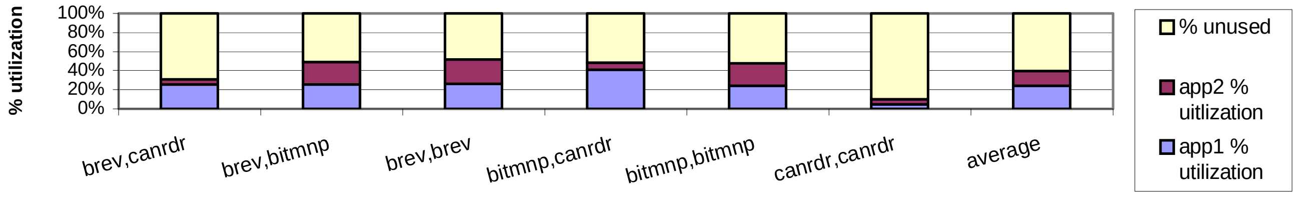

Figure 9: Component utilization for significant (>30% speedup) barrel shifter pairs. The previously shown low speedup overheads are achieved despite the relatively high unit utilizations shown here.

in Section 3.2 for each application relative to a base processor, and averaged for a given pair to obtain a single speedup value.

4.2 Disjunctively-Constrained Knapsack Solution

We determined that the two-processor unit-instantiation problem could be approximately mapped to a variation of the 0-1 knapsack problem. The 0-1 knapsack problem consists of a set of items, with each item having a profit and a weight, and a total weight constraint on the knapsack. The problem is to choose which items to assign to the knapsack such that profit is maximized while not violating the weight constraint.

The key to the problem mapping involves noting that each unit can be considered to be an item in the knapsack problem, with instantiating a unit corresponding to adding the item to the knapsack. *Ignoring conjoinment for the moment*, eight "items" would exist: barrel shifter for processor A, divider for A, multiplier for A, FPU for A, barrel shifter for processor B, divider for B, multiplier for B, and FPU for B. An item's weight would be the corresponding unit's size from Figure 2. An item's profit would be the speedup increment (i.e., the speedup amount above 1.0) that the corresponding unit provides over a base processor for the given application, shown in Figure 6 (e.g., if a multiplier speeds up the application by 1.3x, the profit would be 0.3). This mapping is *approximate* because speedup increments are not necessarily additive – if a barrel shifter provides a speedup of 1.2x, and a multiplier of 1.3x, instantiating both a barrel shifter and multiplier might yield a speedup less than 1.5x, such as 1.4x, due to overlapping functionality (e.g., a multiplier may be used for shifting). However, we examined all pairs of units for all applications, and found that adding speedup increments had an average inaccuracy of 5.9% (though the worst case was 26% between the multiplier and divider), which we considered acceptable.

With the above mapping, we could solve the two-processor unit-instantiation problem using a well-known dynamic programming solution to the 0-1 knapsack problem.

Extending the mapping to consider conjoinment, we can introduce new "items" in addition to those eight listed above: barrel shifter for processors A and B (i.e., one barrel shifter shared by both processors – conjoined), divider for both A and B, multiplier for both A and B, and FPU for both A and B, for twelve items total. However, this extension is not complete, because these items corresponding to conjoined units cannot co-exist in the knapsack with items corresponding to non-conjoined units. For example, we cannot have a multiplier for A, and a multiplier for A and B, both in the solution – either the multiplier is for A, or the multiplier is shared by A and B. Fortunately, the *disjunctively constrained knapsack problem* [13] extends 0-1 knapsack to prohibit certain combinations of items from appearing in the knapsack. We thus specify the prohibited combinations, and apply the algorithm described in [13]. The algorithm is known to be "pseudo-polynomial," and effectively quadratic, proportional to the number of items times the size of the knapsack.

Note that, although the used algorithm optimally solves the knapsack problem, the solution is not necessarily optimal for the two-processor unit-instantiation problem, because the mapping of that problem to knapsack was approximate, due to the additive speedup increment approximation.

5. RESULTS

We obtained solutions for all pairs of our 11 applications (thus, 121 pairs), using our knapsack approach, and using exhaustive search. Due to space limitations, we show results for eight representative pairs, shown in Figure 10. The pairs were selected to show the multiple types of applications. *BaseFP01* and *tblook* both require a floating point unit for large speedups, while *bitmnp* and *canrdr* rely on the barrel shifter and multiplier. With these pairings, we can see how our solution works over the entire search space. We imposed a size constraint guaranteed to "hurt" somewhat by not allowing all units from which an application

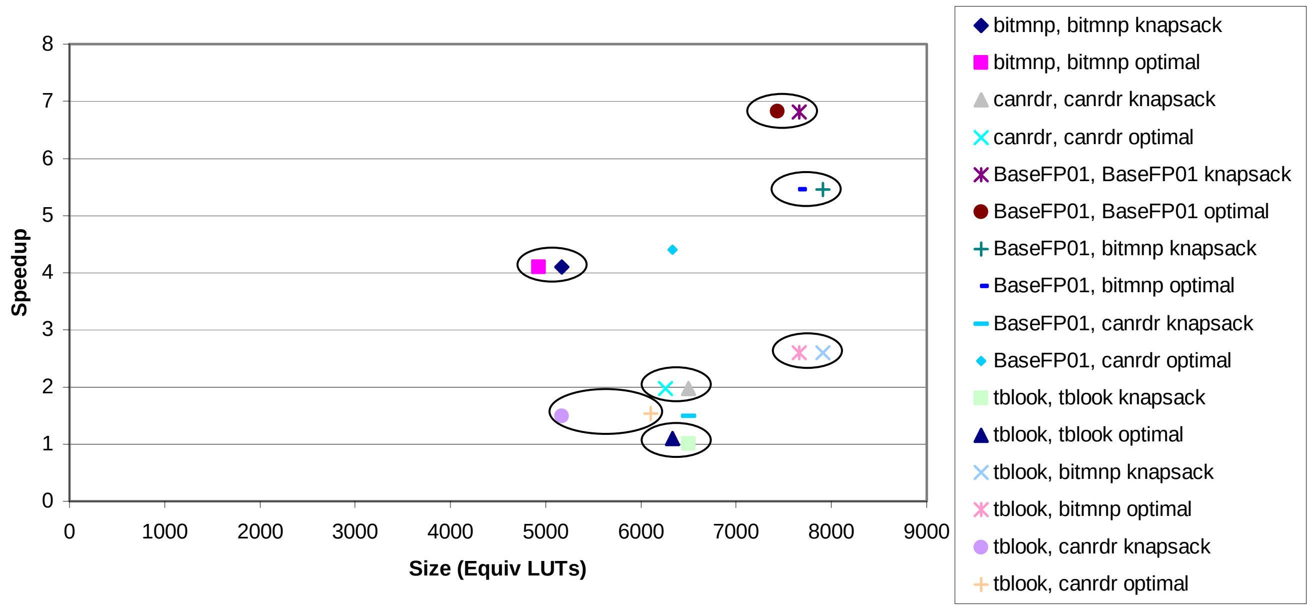

Figure 10: Solutions obtained by exhaustive (optimal) and knapsack algorithms, for eight randomly-selected application pairs, and an area constraint set to 80% of the area of the best configuration. The knapsack algorithm finds near-optimal solutions for seven of the eight pairs (circled), doing poorly on one pair (the two non-circled points).

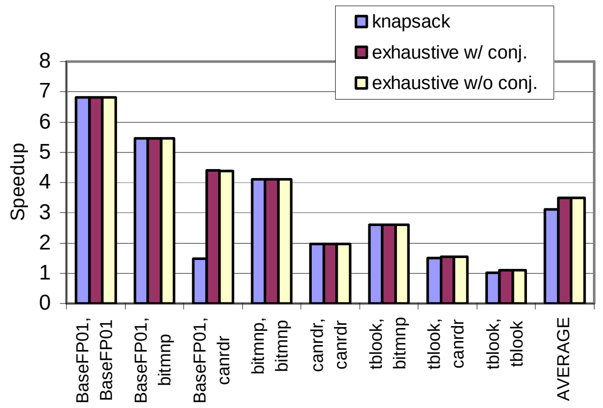

Figure 11: Speedup of the chosen benchmark pairs, using the 80% of the best configuration size constraint.

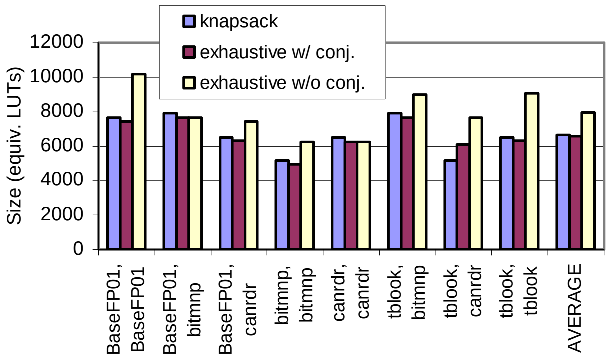

Figure 12: Size reductions for chosen benchmark pairs.

benefits, by first determining for each application what units provide speedup, summing the sizes for each processor having those units, and setting the size constraint to 80% of that size.

Figure 11 shows that the knapsack approach usually obtains near-optimal solutions, sometimes having slightly worse speedup (barely discernible in the figure) and/or slightly worse size (though always satisfying the size constraint, of course). The one sub-optimal case involves the large FPU and occurs due to the additive speedup increment assumption – we plan to investigate improvements to reduce this sub-optimality, where such a solution would likely involve modifications for large units not obeying the additive assumption. Average speedup was within 1% of optimal.

Figure 12 shows the size savings obtained by using conjoinment versus not considering conjoinment. Knapsack achieves nearly the same size savings as exhaustive search, yielding size savings of 16% on average.

Runtimes of the dynamic programming algorithm were under one second in all cases. That time does not include a fixed initial setup time required to obtain size data for each unit through synthesis (requiring tens of minutes), and simulations to determine speedup increments for each unit (requiring seconds).

6. CONCLUSIONS

While customizing soft-core FPGA processors by custom-instantiating datapath units provides for good speedups and efficient size usage, we showed that conjoining processors by sharing those units further reduces size with little impact on speedup. We developed an effective dynamic programming method for automatically finding a good instantiation of units, including conjoined units, for two processors. We showed that considering conjoined units yields 16% average size savings with less than 1% speedup penalty, even using pessimistic performance assumptions.

7. ACKNOWLEFGEMENTS

This research was supported in part by the National Science Foundation (CCR-0203829), by the Semiconductor Research Corporation (2005-HJ-1331), and through hardware and software donations provided by Xilinx.

8. REFERENCES

[1] Abraham, A., B. Rau., Efficent Design Space Exploration in PICO. 2000. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES).

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

[3] Cong, J., Y. Fan, G. Han, Z. Zhang. Application-Specific Instruction Generation for Configurable Processor Architecures. 2004. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).

[4] Cong, J., Y. Fan, G. Han, A. Jagannathan, G. Reinman, Z. Zhang. Instruction Set Extension with Shadow Registers for Configurable Processors. 2005. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).

[5] Dwivedi, B.,A. Kumar, M. Balakrishnan, Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks. 2004. International Symposium on Hardware/Software Codesign and Internation Symposium on System Synthesis (CODES/ISSS).

[6] Huebner, M., T. Becker, J. Becker. Real-Time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration. 2004. 13th Symposium on Integrated Circuit Design and System Design (SBCCI).

[7] Jin. Y., N. Satish, K. Ravindran, K. Keutzer. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems. 2005. International Symposium on Hardware/Software Codesign and Internation Symposium on System Synthesis (CODES/ISSS).

[8] Kumar, R., N. Jouppi, D. Tullsen. Conjoined-core Chip Multiprocessing. In the Proceedings of the 37th International Symposium on Microarchitecture.

[9] Kumar, R., V. Zyuban, D. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. 2005. In the Preceedings of the 32nd International Symposium on Computer Architecture.

[10] Poseidon Triton System. http://www.poseidon-systems.com

[11] Tencillica, www.tencillica.com

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

[13] 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.

[14] Yiannacouras, P., J. Rose, J. Steffan, The Microarchitecture of FPGA-Based Soft Processors. 2005. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES).

	aifir	BaseFP01	brev	bitmnp	canrdr	g3fax	g721_ps	matmul	raytrace	tblook	ttsprk
ttsprk	bs: 0.84% div: mul: 1.01% fpu:	bs: 0.21% div: mul: 0.34% fpu:	bs: 5.82% div: mul: 1.40% fpu:	bs: 5.71% div: mul: 0.78% fpu:	bs: 1.45% div: mul: 1.63% fpu:	bs: 0.71% div: mul: 0.33% fpu:	bs: 1.51% div: mul: 0.60% fpu:	bs: 0.87% div: mul: 1.08% fpu:	bs: 0.33% div: mul: 0.33% fpu:	bs: 0.36% div: mul: 1.11% fpu:	bs: 0.56% div: 1.21% mul: 1.11% fpu:
tblook	bs: 0.80% div: mul: 0.80% fpu:	bs: 0.16% div: mul: 0.17% fpu: 0.27%	bs: 5.76% div: mul: 1.20% fpu:	bs: 5.70% div: mul: 0.59% fpu:	bs: 1.41% div: mul: 1.41% fpu:	bs: 0.67% div: mul: 0.16% fpu:	bs: 1.47% div: mul: 0.42% fpu:	bs: 0.82% div: mul: 0.89% fpu:	bs: 0.28% div: mul: 0.16% fpu: 0.10%	bs: 0.31% div: mul: 0.34% fpu: 0.19%	
raytrace	bs: 0.75% div: mul: 0.61% fpu:	bs: 0.13% div: mul: 0.01% fpu: 0.38%	bs: 5.70% div: mul: 0.98% fpu:	bs: 5.59% div: mul: 0.41% fpu:	bs: 1.36% div: mul: 1.17% fpu:	bs: 0.63% div: mul: 0.00% fpu:	bs: 1.42% div: mul: 0.24% fpu:	bs: 0.79% div: mul: 0.68% fpu:	bs: 0.47% div: mul: 0.00% fpu: 0.73%		
matmul	bs: 1.32% div: mul: 1.44% fpu:	bs: 0.66% div: mul: 0.69% fpu:	bs: 6.87% div: mul: 1.79% fpu:	bs: 6.58% div: mul: 1.17% fpu:	bs: 2.00% div: mul: 2.09% fpu:	bs: 1.21% div: mul: 0.68% fpu:	bs: 2.07% div: mul: 0.98% fpu:	bs: 1.53% div: mul: 1.35% fpu:			
g721_ps	bs: 1.16% div: mul: 0.90% fpu:	bs: 0.51% div: mul: 0.25% fpu:	bs: 6.42% div: mul: 1.30% fpu:	bs: 6.26% div: mul: 0.69% fpu:	bs: 1.80% div: mul: 1.51% fpu:	bs: 1.87% div: mul: 0.25% fpu:	bs: 4.03% div: mul: 0.48% fpu:				
g3fax	bs: 1.16% div: mul: 0.61% fpu:	bs: 0.51% div: mul: 0.01% fpu:	bs: 6.42% div: mul: 0.98% fpu:	bs: 6.26% div: mul: 0.98% fpu:	bs: 1.80% div: mul: 0.41% fpu:	bs: 0.99% div: mul: 0.01% fpu:					
canrdr	bs: 1.95% div: mul: 2.02% fpu:	bs: 1.21% div: mul: 1.18% fpu:	bs: 7.92% div: mul: 2.58% fpu:	bs: 7.60% div: mul: 1.74% fpu:	bs: 2.41% div: mul: 2.34% fpu:						
bitmnp	bs: 6.55% div: mul: 1.10% fpu:	bs: 5.32% div: mul: 0.42% fpu:	bs: 16.51% div: mul: 1.53% fpu:	bs: 13.10% div: mul: 0.85% fpu:							
brev	bs: 6.65% div: mul: 1.82% fpu:	bs: 5.39% div: mul: 0.98% fpu:	bs: 16.97% div: mul: 1.95% fpu:								
BaseFP01	bs: 0.62% div: mul: 0.62% fpu:	bs: 0.02% div: mul: 0.02% fpu: 0.57%									
aifir	bs: 1.27% div: mul: 1.21% fpu:										

Figure 13: Performance overhead for conjoined units for all pairs of applications. The % shown is the performance overhead caused by stalls due to conjoined unit contention and by 1 extra cycle for accessing a conjoined unit (even without contention), versus using non-conjoined units (with no extra cycle for access). The overheads are the average of the two overheads of the applications in a pair.