Two-Level Microprocessor-Accelerator Partitioning

Scott Sirowy, Yonghui Wu, Stefano Lonardi, Frank Vahid*

Department of Computer Science and Engineering – University of California, Riverside
{ssirowy, yonghui,stelo, vahid}@cs.ucr.edu

*Also with the Center for Embedded Computer Systems, University of California, Irvine

Abstract

The integration of microprocessors and field-programmable gate array (FPGA) fabric on a single chip increases both the utility and necessity of tools that automatically move software functions from the microprocessor to accelerators on the FPGA to improve performance or energy. Such hardware/software partitioning for modern FPGAs involves the problem of partitioning functions among two levels of accelerator groups – tightly-coupled accelerators that have fast single-clock-cycle memory access to the microprocessor's memory, and loosely-coupled accelerators that access memory through a bridge to avoid slowing the main clock period with their longer critical paths. We introduce this new two-level accelerator-partitioning problem, and we describe a novel optimal dynamic programming algorithm to solve the problem. By making use of the size constraint imposed by FPGAs, the algorithm has what is effectively quadratic runtime complexity, running in just a few seconds for examples with up to 25 accelerators, obtaining an average performance improvement of 35% compared to a traditional single-level bus architecture.

1. Introduction

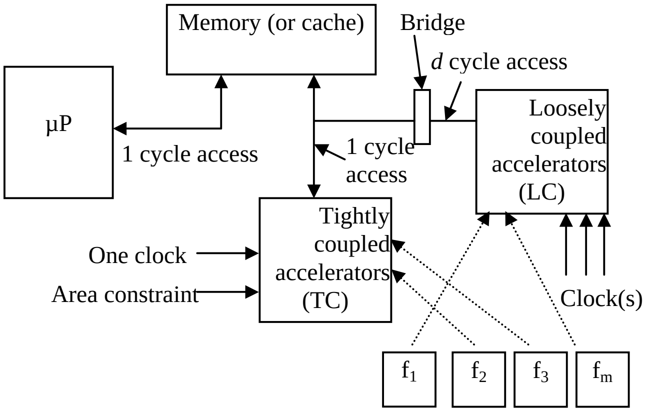
Platforms incorporating both a microprocessor and FPGA (Field-Programmable Gate Array) fabric on a single chip are becoming an increasingly popular software implementation platform in embedded computing systems. Some such platforms include hard-core processors, which are physically designed onto the chip alongside the FPGA fabric. Other platforms utilize soft-core processors, which are synthesized onto the FPGA fabric itself. Incorporating both a microprocessor (hard or soft core) and FPGA fabric on a single chip provides several advantages over multi-chip solutions, including reduced part counts, faster communication between the microprocessor and the logic mapped to the FPGA, and potentially reduced system costs.

The close proximity of FPGA fabric to a microprocessor encourages movement of a microprocessor program's critical computations from microprocessor execution to custom processor circuit execution on FPGA fabric, to obtain substantial speedups ranging from 2x to 100x, as well as energy savings [1][5] [10][18]. Such hardware/software partitioning takes two forms, one multi-processing oriented, the other sequential processing oriented. The multi-processing oriented form, sometimes referred to as system synthesis, maps a task graph to a set of concurrently-executing communicating microprocessors and custom processors [4][11]. The sequential processing oriented form creates custom circuits to execute commonly-executed functions (or sequences of instructions) found in a single sequential program of one microprocessor [8][10][15][18]. Several commercial tools supporting the sequential processing form of partitioning have recently appeared [3][16]. ASIP (application-specific instruction-set processing) approaches[21] may also be viewed as a sequential form of partitioning.

We focus on the sequential form of partitioning. In that form, the custom circuits may be viewed as *accelerators*, standard forms of which include floating point accelerators and graphics accelerators. In stark contrast to the multi-processing form of partitioning in which processors execute concurrently and contend for resources, the accelerators in the sequential processing form typically execute as microprocessor slaves, thus greatly simplifying communication and synchronization issues.

Previous partitioning work has assumed a single clock frequency for all of a microprocessor's accelerators, or ignores clock frequencies entirely. However, modern FPGA technologies support the use of dozens of different clock frequencies on a single device. Thus, a new aspect of the partitioning problem consists of determining which accelerators should be tightly-coupled to the microprocessor, and which should be loosely-coupled. *Tightly-coupled accelerators* have direct access to the microprocessor memory or cache, and thus should operate at a single clock frequency, which will necessarily be the lowest frequency of any of those accelerators. *Loosely-coupled accelerators* instead access the memory through a bridge, and thus may have individually optimized clock frequencies. For example, a Xilinx Microblaze soft-core processor utilizes a dual-port block RAM for memory (or cache), as shown in Figure 1. Tightly-coupled accelerators and a bridge access the second port of that RAM using a single frequency. Mapping a given function to the tightly-coupled group provides single-cycle access but at the expense of running at a possibly slower frequency, versus mapping to the loosely-coupled group to run at the fastest possible individual frequency, but requiring multiple cycles through the bridge for memory accesses. Thus, a new partitioning problem exists that seeks to determine the best mapping of functions among tightly-coupled and loosely-coupled groups to achieve best overall performance – a problem we refer to as the

Figure 1: Target two-level coupled architecture, derived from Xilinx's Microblaze base architecture



978-3-9810801-2-4/DATE07 © 2007 EDAA

Figure 3: Complete two-level accelerator partition solution space for a 10-accelerator example, showing the benefit of finding the best two-level accelerator partition vs. making all processors tightly-coupled or all processors loosely-coupled.

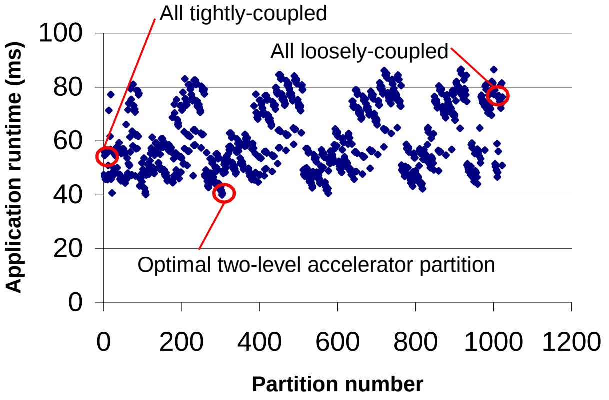
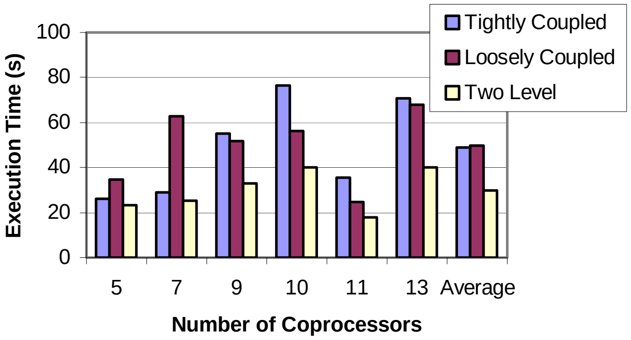


Figure 2: The need for a two level coupling architecture. In all applications examined, two-level accelerator partitioning resulted in improvement, sometimes quite substantial.



two-level accelerator partitioning problem.

Although modern FPGA architectures motivated our work on the problem, as multiple clock domains are becoming common in ASIC technology also [1], the problem may therefore also exist for ASIC microprocessor/accelerator architectures supporting multiple clock domains.

Figure 3 illustrates the benefits of considering two-levels of accelerators under the above clock frequency constraints, for an application with 10 accelerators, with the 1024 possible partitionings along the x-axis, and the application's runtime on the y-axis. The figure shows that making all accelerators either tightly-coupled or loosely-coupled results in significantly slower performance than the best two-level partition. Figure 2 further highlights that a two-level partitioning of an application with varying numbers of accelerators results in superior execution times over partitioning all accelerators tightly or all loosely.

Previous hardware/software partitioning work does not consider two-levels of coupling and in particular does not consider the clock frequency interactions among tightly-coupled accelerators, instead assuming all accelerators have single-cycle access [9][15][17][18], assuming all accelerators have multiple cycle access [12][20], or simply associating an execution time with functions without considering details of clock frequency [4][10].

We present two contributions in this paper. We define the two-level accelerator partitioning problem and show the performance benefits achievable by solving the problem. Also, we introduce a fast optimal algorithm that scales well for even large problem sizes. The key to the algorithm is to map the problem to a series of 0-1 knapsack problems, and then to solve each knapsack problem using a dynamic programming solution having a pseudo-polynomial runtime, resulting in polynomial (effectively quadratic) runtime. A fast optimal algorithm not only ensures the best results, but potentially enables repeated use of the algorithm as part of higher-level exploration approaches without accumulated decision errors due to sub-optimal partial solutions. To our knowledge, our algorithm is novel and might be applied to number of different problems not previously explored. Many modern systems implement a two-level bus structure very similar to the architecture in Figure 1, and thus our problem solution is applicable to many systems that require performance gains.

2. Two-level Accelerator Partitioning Problem Definition

The two-level accelerator partitioning problem takes as input a set of functions to be implemented as accelerators, determined by a previous hardware/software partitioning decision (note that our problem and hence algorithm may actually be a sub-problem of a higher-level exploration technique, and thus hardware/software partitioning and two-level accelerator partitioning may be done iteratively). Each accelerator is annotated with four numbers, determined from the synthesized circuit generated for the function: the number of memory accesses, the total number of computation cycles, the synthesized area, and the maximum possible clock frequency. These numbers are straightforwardly obtained using simulation and synthesis [18][19]. The number of memory accesses and computation cycles may represent averages or worst-case numbers, depending on whether the designer seeks to optimize for overall average or worst-case performance.

The number of extra cycles introduced by the bridge is also given. This memory access penalty is an architectural feature of the bridge, and not a per-application number, so the number is fixed for all applications. A loosely-coupled accelerator would incur this latency penalty each time it made an access to memory, since the accelerator is connected to the memory through the bridge.

All tightly-coupled accelerators, having single-cycle access to memory or cache, must run at a single clock frequency – this assumption matches several modern commercial FPGAs that incorporate microprocessors. Because all those accelerators must run at one clock frequency, they all must run at the frequency of the *slowest* tightly-coupled accelerator in the group. The tightly-coupled accelerators' frequency need not be the same as the microprocessor's frequency. Loosely-coupled accelerators, in contrast, each run at their unique fastest clock frequency. Modern FPGAs support multiple clock frequencies on the same platform. For instance, the Xilinx Virtex II Pro supports eight unique clock frequencies, and the trend is towards more frequencies per device.

Formally, the problem takes as input a set F of *n* functions {f_0, f_1, … f_n}, and each function requires accelerator implementation. Each function *f* has four attributes: f_i*.comp_cycles*, f_i*.mem_accesses*, f_i*.clk_freq*, and f_i*.area*. The problem definition involves two initially empty sets, *TC*, which represents tightly-coupled accelerators, and *LC*, which represents loosely-coupled accelerators. Each function in *F* must be mapped to exactly one of the sets *TC* or *LC*.

The objective function is to minimize the total execution time of all the accelerators, computed as follows:

$$TC([\sum_{i=1}^{n}(comp\_cycles_i + mem\_accesses_i)] / \min\_clock)$$

$$+ LC(d * \sum (mem\_accesses_i / clk\_freq_i)$$

$$+ \sum_{i=1}^{n}(comp\_cycles_i) / clk\_freq_i))$$

min_clock is the minimum clock frequency within the *TC* set. *d* is the memory delay from the bridge for the loosely coupled accelerators. Figure 1 showed a sample architecture and mapping. The architecture is based on standard two-level architectures with both a local processor bus and peripheral bus, both of which have access to a shared memory (cache).

A size constraint exists for the tightly-coupled accelerator group, due to FPGA congestion issues relating to providing multiple accelerators with single-cycle access to memory or cache. In the completely performance-driven problem where no such constraint exists, we simply utilize a constraint larger than all accelerators to match our formulation. No size constraint exists for the loosely-coupled accelerators, as we assume that the previous hardware/software partitioning ensured that the functions mapped to accelerators fit on available FPGA resources. However, in the case where a size constraint does exist for the loosely-coupled accelerators, a second FPGA could be added, which would also communicate through a bridge.

The above problem definition has the limitation of not considering the situation where the number of frequencies available to the loosely-coupled processors is less than the number of such processors. We plan to consider that situation, along with architectures having more than two levels of accelerators, in future work.

3. NKDP Algorithm- *N* 0-1 Knapsacks and Dynamic Programming

3.1 Exhaustive and greedy solutions

To solve the above problem, we first developed an *exhaustive* search algorithm. Exhaustive search finds the optimal solution in a few seconds for problems involving up to about 15 accelerators. Larger problems require minutes or hours, and the algorithm does not complete in any reasonable time for problems larger than 20 functions.

We also developed a *greedy* heuristic. The heuristic starts with all functions mapped to the loosely-coupled group. It orders the functions according to their contribution to total execution time. It then considers each function in that order, and moves a function to the tightly-coupled group if such a move improves the application runtime and if the function fits in the remaining available size of the tightly-coupled group. This heuristic is fast, but we found that the heuristic yielded solutions 15% worse on average compared to optimal.

3.2 NKDP solution

We sought to develop a solution that would yield closer-to-optimal solutions in reasonable runtime. Upon investigating such a solution, we came upon an idea that would actually yield optimal solutions, yet in effectively polynomial time. (The partitioning problem is known to be NP-complete [14], so a truly polynomial-time solution is not possible.)

Figure 4: NKDP algorithm

```
NKDP (S, n, d, A)
  1. A← Sort the accelerators in A in the decreasing order of their frequencies
  2. min_t ← $\sum_{i=1}^{n}(A[i].mem * f + A[i].c\_c) / A[i].clk$
  3. opt_sol ← {∅}
  4. for i ← 1 to n
      4.1. freq = A[i].fq
      4.2. for j ← 1 to (i-1)
          V[j] ← ((A[j].mc * f + A[j].cc) / A[j].fq) – ((A[j].mc + A[j].cc) /
              freq);
          W[j] ← A[j].size;
  4.3. S’ = S – A[i].size
  4.4. tmp_sol,  tmp_t ← Knapsack01(V,W,S’)
  4.5. tmp_sol ← tmp_sol ∪ A[i];  tmp_t ← tmp_t + (A[i].mc + A[i].cc)/ freq
  4.6. if  tmp_t < min_t
          min_t ← tmp_t, opt_sol ← tmp_sol
  5. return opt_sol
```

The key idea to our solution approach is that the two-level accelerator partitioning problem with *n* functions can be decomposed into *n* 0-1 knapsack problems. In the classic 0-1 knapsack problem, the goal is to choose a subset of the items whose total value is maximized while at the same time the sum of the weights does not violate the constraint on the overall capacity given the value and the weight of *n* items to be stored, and the capacity of the knapsack *S*. This problem is NP-complete, but can be solved optimally with a dynamic programming approach in pseudo-polynomial time.

We refer to our solution as the *n-knapsack dynamic programming*, or ***NKDP***, solution. The pseudo code is presented in Figure 4. The inputs to our algorithms are *S*: the total area constraint of the tightly coupled group, *n*: the number of accelerators, *d*: memory access penalty for the bridge, and *A*: an array of *n* accelerators. The output from the algorithm is the optimal set of accelerators to be tightly coupled.

To the best of our knowledge, our solution approach to the two-level accelerator partitioning problem is novel. The idea behind our algorithm is that if we “would know” the slowest accelerator in the tightly-coupled set (let the accelerator be X), we can optimally map all the functions to the tightly and loosely coupled sets as follows:

1) Map X to the tightly-coupled set, since based on our assumption, X is in the tightly-coupled set.
2) Map all functions whose accelerators have a *slower* frequency than X to the loosely-coupled set, because otherwise mapping that function to the tightly-coupled set would violate our assumption that X is the slowest accelerator in the tightly-coupled set.
3) Let the set of functions whose accelerators have the same frequencies as or higher frequencies than X be the set S_FAST. For each function in S_FAST, calculate the reduction in the function’s execution time should that function be mapped to the tightly-coupled set as opposed to the loosely-coupled set. This calculation can be done because the function’s execution time as a tightly-coupled accelerator is known (because we know the function will run at the same

frequency as that of X), and because the function's execution time as a loosely-coupled processor is known (because we know the function's accelerator clock frequency and the memory access penalty). Note that the reduction in execution time can be negative, which means mapping the function to the tightly-coupled set will lengthen its execution time. If that happens, the function is mapped to the loosely-coupled set immediately, and is removed from S_FAST.

4) Now the problem of mapping the functions in set S_FAST is reduced to the classic 0-1 knapsack problem, where S_FAST contains the set of items to be chosen, the weight of each item is just the size of the corresponding accelerator, the value of each item is the reduction of the function's execution time that was calculated in the previous step, and the capacity of the knapsack is the area constraint of the overall tightly coupled group minus the area of X. We seek a subset of S_FAST that maximizes the overall reduction in the execution time while still satisfying the total area constraint of the tightly-coupled set.

5) The 0-1 knapsack problem that is induced in the previous step can be solved optimally by dynamic programming, as we showed in line 4.2 in the pseudo code, which has a time complexity of *O(Sn)*, where *n* is the number of items in set S_FAST and *S* is the capacity of the knapsack. The optimum solution to the above 0-1 knapsack problem corresponds to the sub-set of accelerators in S_FAST that should be mapped as tightly coupled. The rest of the accelerators should all be mapped as loosely coupled.

The above steps will yield the optimum solution if X is known. Of course, we do not know X in advance, but that does not matter since we can try all the possible choices of X. For each function, we assume the function is X, and we run the above five steps to obtain a locally-optimal solution. Among all the locally-optimal solutions thus obtained, the one that has the minimum overall execution time must be globally optimal.

In our earlier algorithm pseudo-code, we first sort the functions in decreasing order of their frequencies (line 2 of the pseudo-code), such that the set S_FAST that corresponds to the current choice of X can be easily identified, which are just the functions that precede X in the list.

3.3 NKDP complexity

Since our algorithm decomposes the original problem into n 0-1 knapsack problems, and solves each optimally via dynamic programming (which has a time complexity of O(*Sn*) as we mentioned earlier), the overall time complexity becomes $O(Sn^2)$. In practical applications, the number of functions n will rarely be higher than fifty, while the size of the knapsack S will usually be on the order of thousands (of combinational logic blocks or lookup tables) for FPGA technology and typical numbers of functions mapped to accelerators. These figures allow us to claim that our algorithm is in practice computationally efficient, and at the same time the solution it computes is globally optimal.

3.4 NKDP Quantization

We observe that our dynamic programming formulation relies on the value of the area constraint input in order to achieve fast algorithm runtimes. We briefly mentioned in the last section that the area constraints of typical FPGAs are on the order of thousands, which could potentially make the dynamic programming algorithm run very slow. The steady increase in the amount of configurable logic on typical FPGAs exacerbates the situation. However, since we are mainly concerned with application runtime, and area constraints are usually a soft constraint, especially in the early design stage when such a decision is made, we can reduce the area constraint by dividing by a quantization factor, and still achieve near-optimal configuration, as long as we quantize the areas of the accelerators by the same factor. Quantizing the area inputs by a factor of ten would result in filling in a dynamic programming table one-tenth its original size; quantizing by 100 would yield a table one-hundredth its original size. Such optimizations would result approximately in algorithm speedups of 10x and 100x. The proposed quantization technique makes NKDP suitable for an even larger design space exploration and/or a dynamically tuned environment.

4. Experiments and Results

This section presents results of applying the NKDP algorithm to a standard benchmark, a commercial quality H.264 video decoder, as well as to several synthetic examples.

To evaluate both the quality and performance of our algorithm, we implemented NKDP, greedy, and exhaustive solutions. We wrote our implementation with several hundred lines of C. We ran our experiments on an Intel Celeron 2.5 GHz machine running with 512 MB RAM.

We first examine NKDP using various levels of quantization, particularly to determine the returns we achieve for larger quantization factors, and whether or not those have any effect on the solution found. Figure 5 shows our findings from applying quantization factors to NKDP of ten and hundred to three applications. We observe from Figure 5(a) that moving from no quantization to a factor of ten results in a much larger difference in algorithm execution time than moving from ten to hundred. This suggests there are diminishing returns in applying quantization to NKDP, at the risk of altering the area constraint too much. We recall that for every order of magnitude of area

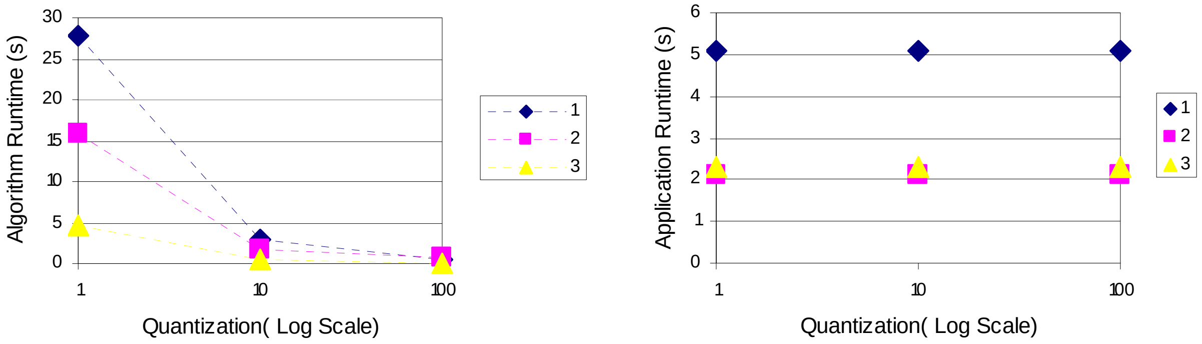
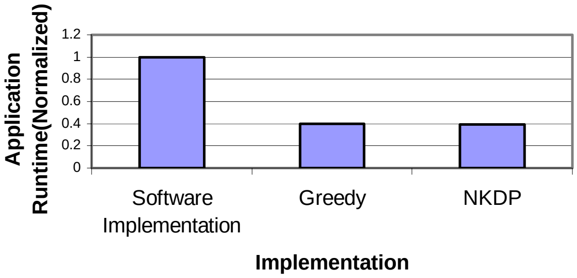


Figure 5: Quantizing area on three applications (labeled 1, 2, and 3) to improve NKDP algorithm runtime: (a) Algorithm runtime for quantization factors of 10 and 100, and (b) resulting application runtimes found by the algorithm. Quantization factors of 10 and 100 improve runtimes with little impact on quality of results.

Figure 6: Pegwit partitioned functions for accelerator generation.

Function	Compute Cycles	Memory Accesses	Clock Freq. (MHz)	Area (LUTs)
1	405	645	61	618
2	394	627	74	411
3	44	70	40	273
4	44	70	50	305

Figure 8: Pegwit benchmark results showing that NKDP finds the optimal solution with an application runtime reduction of 60%.



quantized, we lose that much accuracy in how much area was actually used for the tightly couple accelerator group. Figure 5(b) shows both quantization factors still achieve the same application execution time, but this may not be the case if we start to quantize too much. Through similar experiments with other benchmarks, we felt a quantization factor of ten achieves both an optimal partition as well as a fast algorithm runtime, suitable for even the largest of partitioning problems.

We next begin examining a benchmark derived from the Pegwit decoder benchmark of MediaBench [13]. Figure 6 shows, for the most critical four functions of the benchmark, the compute cycles, number of memory accesses, clock frequency, and area for the accelerators that would be synthesized for each function. We obtained these figures from both a Xilinx synthesis tool and hand analysis of the Pegwit C code.

Figure 8 shows the benchmark execution time achieved by partitioning using the NKDP solution, compared with a software-only solution. The greedy solution is also shown. While the greedy solution was also able to find the optimal for this benchmark, it fails to do so in later examples. Both NKDP and the greedy heuristic partitioned functions one and two as tightly coupled accelerators, and functions three and four as loosely coupled. This partitioning makes sense, as partitioning either function three or four would result in a large clock penalty on the tightly coupled set. Also, since functions one and two spend a significant portion of their time accessing the memory, partitioning either function loosely would have resulted in a large latency penalty through the bridge to the memory (cache). Figure 7 further expands our findings on Pegwit to show that the two-level partitioning of the accelerators results in a superior execution time over mapping all functions tightly or all loosely.

Figure 9 shows the partitioning results obtained by the NKDP algorithm under different area constraints imposed on the tightly-coupled accelerator set. The results indicate that the algorithm readily handles a variety of area constraints. The optimal partitioning of the four accelerators for Pegwit with no size constraint would be to tightly couple functions one and two, and loosely couple functions three and four. However, when we introduce an area of constraint of one thousand LUTs for the tightly coupled accelerators, we observe that the optimal solution is to tightly couple functions one and four, and loosely couple the other two. If we restrict the Pegwit decoder to only 750 LUTs, then tightly coupling only function one yields the best mapping. It would have been possible to tightly couple functions three and four instead, but because coupling function three tightly would have resulted in a clock speed of 40 MHz, the overall application runtime would have taken a significant hit. Finally, if we restrict the decoder to 600 LUTs, tightly coupling only function two becomes the only real choice since it is the only significant accelerator to be able to fit within the constraint.

Figure 7: Comparing NKDP to all tightly coupled, all loosely coupled, optimal (exhaustive), and greedy heuristic.

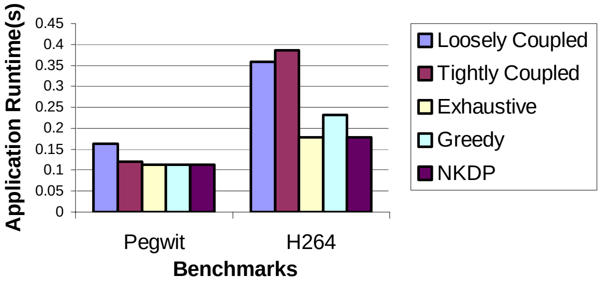
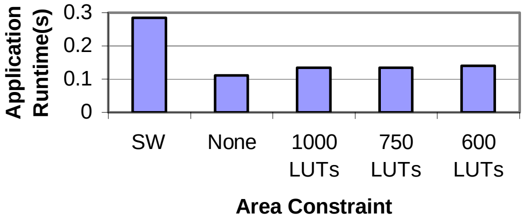


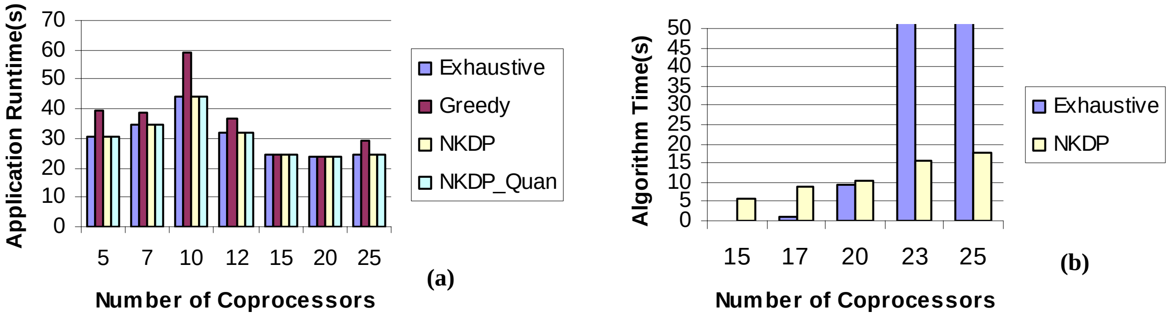
Figure 9: Pegwit benchmark partitioning with different area constraints on the set of tightly-coupled accelerators, compared to a full software implementation



We also tested our two-level partitioning algorithm on a proprietary H.264 video decoder, part of the MPEG-4 standard. H.264 consists of a number of functions suitable for hardware implementation. In our experiments, we chose to implement the eight most critical functions through results we obtained from profiling and synthesis. The functions are primarily targeted at the frame conversion stage of the decoding process, and supplied ample opportunity for hardware acceleration. Through synthesis and hand analysis of the C code, we were able to extract estimates of the number of computation cycles, memory accesses, clock frequency, and area associated with each function. The results of running our greedy heuristic and NKDP algorithm are also shown in Figure 7. An exhaustive search is shown for comparison purposes to show that NKDP was able to find the optimal configuration. We also show that a two-level partitioning results in almost half the execution time than mapping all functions tightly or all loosely. For the eight functions we looked at, NKDP partitioned seven of the functions tightly and one loosely. Closer inspection of the functions' respective clock frequencies revealed that the seven functions coupled tightly all had very similar clock frequencies, while the loosely coupled accelerator had a much lower frequency. Such a mapping is intuitive since the seven coupled tightly together would not incur a large penalty because their frequencies were similar, while the eighth function would have caused the group to suffer a large clock frequency penalty.

We present results for seven different synthetic benchmarks of increasing numbers of functions, shown in Figure 10. Figure

Figure 10: Results for applications of increasing numbers of functions, comparing exhaustive, greedy, NKDP, and NKDP Quantized solutions: (a) application runtimes, (b) algorithm runtimes (greedy and NKDP Quantized were less than one second and are thus not shown).



10(a) shows that NKDP and a quantized version achieves the optimal results, verified by exhaustive search, as expected because NKDP is designed to find the optimal. That figure also shows that the greedy heuristic defined earlier sometimes does not find the optimal, and in a few cases is significantly inferior to the optimal. Figure 10(b) shows that the NKDP runtime scales quite reasonably with problem size, unlike exhaustive search, whose exponential growth becomes evident at around 20 functions. Furthermore, NKDP_quantized runs in under 1 second even for 25 functions, with no change in solution quality.

5. Conclusions and Future Work

We introduced the two-level accelerator partitioning problem, and presented a novel and efficient solution, NKDP, based on a decomposition of the problem into a series of 0-1 knapsack problems. NKDP has pseudo-polynomial runtime, and executes in just seconds for practical-sized examples. The solution produced optimal two-level partitions outperforming a single level accelerator architecture by an average of 35%, and outperforming a greedy two-level partitioning heuristic by an average of 15%. We also showed that quantizing accelerator sizes could yield more than 20x algorithm runtime improvements with no noticeable degradation of partition quality, yielding algorithm runtimes under 1 second for even large examples. We plan to extend our techniques to consider more complex exploration spaces, such as considering a finite number of clock frequencies, multiple frequencies for tightly-coupled processors, multidimensional resource constraints that consider hard-core resources like multipliers and block RAMs, handling memory accesses that don’t all take the same amount of time, and architecture with more than two levels of accelerators.

6. Acknowledgements

This work was supported in part by the National Science Foundation (CNS-0614957) and the Semiconductor Research Corporation (2005-HJ-1331), and by donations from Xilinx Corp.

References

[1] Chattopadhyay, A. and Z. Zilic. GALDS: A Complete Framework for Designing Multiclock ASICs and SoCs. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, Vol. 13, No. 6, June 2005

[2] Compton, K. and S. Hauck. Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. 34, 2 (Jun. 2002), 171-210. 2002

[3] CriticalBlue. http://www.criticalblue.com

[4] Eles, P., Z. Peng, K. Kuchcinsky, and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. Design Automation for Embedded Systems, vol2, no 1, 5-32 January 1997.

[5] Excalibur. Altera Corp., http://www.altera.com

[6] Galanis, M.D, A. Milidonis, G. Theodoridis, D. Soudris,and C. E. Goutis. A Partitioning Methodology for Accelerating Applications in Hybrid Reconfigurable Platforms. Design Automation and Test in Europe (DATE), pp. 247-252, 2005.

[7] Guo, Z., W. Najjar, F. Vahid, and K. Vissers.. A quantitative analysis of the speedup factors of FPGAs over processors. In Proceedings of the 2004 ACM/SIGDA 12th international Symposium on Field Programmable Gate Arrays. FPGA '04. ACM Press, New York, NY, 162-170.2004

[8] Gupta, R. and G. De Micheli. Hardware-Software Cosynthesis For Digital Systems. IEEE Design and Test of Computers. Pages 29-41, September 1993.

[9] Hauser, J.R. and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Accelerator. FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on16-18 April 1997 Page(s):12 – 21

[10] Henkel, J. A low power hardware/software partitioning approach for core-based embedded systems. In Proceedings of the 36th ACM/IEEE Design Automation Conference, 122–127.1999

[11] Kalavade, A. and Subrahmanyam, P. A. 1997. Hardware/software partitioning for multi-function systems. In *Proceedings of the 1997 IEEE/ACM international Conference on Computer-Aided Design*

[12] Laufer, R., R.R Taylor, and H. Schmit..PCI-Piperench amd the Sword API: A system for Stream-based Reconfigurable Computing. Field-Programmable Custom Computing Machines. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on 21-23 April 1999 Page(s):200 – 208.1999

[13] Lee, C., M. Potkonjak., and W.H Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In Proceedings of the 30th Annual ACM/IEEE international Symposium on Microarchitecture International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 330-335. 1997.

[14] Lengauer, T. 1990. Combinatorial Algorithms for Integrated Circuit Layout. John Wiley & Sons, Inc., New York, NY.

[15] Miyamori, T., and U. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. FPGAs for Custom Computing Machines. Proceedings. IEEE Symposium on 15-17 April 1998 Page(s):2 – 11.1998

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

[17] Rupp, C.R.; M. Landguth,, T. Garverick., E. Gomersall, H. Holt.; J.M Arnold,. And M. Gokhale. The NAPA Adaptive Processing Architecture. FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on15-17 April 1998 Page(s):28 - 37

[18] Stitt, G., F. Vahid, and S. Nematbakshi. Energy Savings and Speedups From Partitioning Critical Software Loops to Hardware in Embedded Systems. IEEE Transactions on Embedded Computer Systems, January 2004.

[19] Suresh, D. C., W.A Najjar,, F. Vahid,., J. Villarreal., and G. Stitt.. Profiling tools for hardware/software partitioning of embedded applications. In Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool For Embedded Systems (San Diego, California, USA, June 11 - 13, 2003). LCTES '03. ACM Press, New York, NY, 189-198. 2003.

[20] Wildfire Reference Manual, Annapolis, Maryland: Annapolis Microsystems, Inc., 1998

[21] Yiannacouras, P., Steffan, J. G., and Rose, J. 2006. Application-specific customization of soft processor microarchitecture. In *Proceedings of the internation Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 22 - 24, 2006). FPGA'06.