CHAPTER 26

Hardware/Software Partitioning

Frank Vahid, Greg Stitt
Department of Computer Science and Engineering
University of California–Riverside

Field-programmable gate arrays (FPGAs) excel at implementing applications as highly parallel custom circuits, thus yielding fast performance. However, large applications implemented on a microprocessor may be more size efficient and require less designer effort, at the expense of slower performance. In some cases, mapping an entire application to a microprocessor satisfies performance requirements and so is preferred. In other cases, mapping an application entirely to custom circuits on FPGAs may be necessary to meet performance requirements. In many cases, though, the best implementation lies somewhere between these two extremes.

Hardware/software partitioning, illustrated in Figure 26.1, is the process of dividing an application between a microprocessor component ("software") and one or more custom coprocessor components ("hardware") to achieve an implementation that best satisfies requirements of performance, size, designer effort, and other metrics.[1] A custom coprocessor is a processing circuit that is tailor-made to execute critical application computations far faster than if those computations had been executed on a microprocessor.

FPGA technology encourages hardware/software (HW/SW) partitioning by simplifying the job of implementing custom coprocessors, which can be done just by downloading bits onto an FPGA rather than by manufacturing a new integrated circuit or by wiring a printed-circuit board. In fact, new FPGAs even support integration of microprocessors within an FPGA itself, either as separate physical components alongside the FPGA fabric ("hard-core microprocessors") or as circuits mapped onto the FPGA fabric just like any other circuit ("soft-core microprocessors"). High-end computers have also begun integrating microprocessors and FPGAs on boards, allowing application designers to make use of both resources when implementing applications.

Hardware/software partitioning is a hard problem in part because of the large number of possible partitions. In its simplest form, hardware/software partitioning considers an application as comprising a set of *regions* and maps

[1] The terms *software*, to represent microprocessor implementation, and *hardware*, to represent coprocessor implementation, are common and so appear in this chapter. However, when implemented on FPGAs, coprocessors are actually just as "soft" as programs implemented on a microprocessor, with both consisting merely of a sequence of bits downloaded into a physical device, leading to a broader concept of "software."

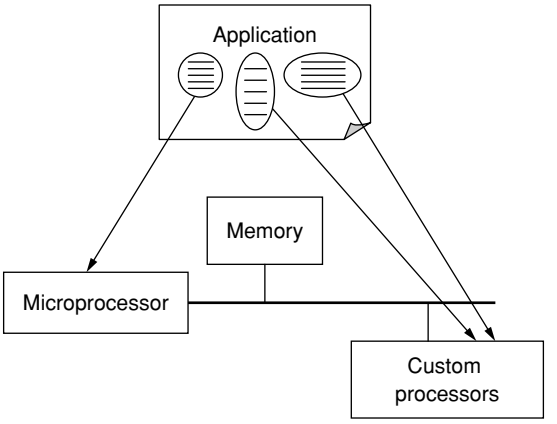

FIGURE 26.1 ■ A diagram of hardware/software partitioning, which divides an application between a microprocessor component ("software") and custom processor components ("hardware").

each region to either software or hardware such that some cost criteria (e.g., performance) is optimized while some constraints (e.g., size) are satisfied.

A *partition* is a complete mapping of every region to either hardware or software. Even in this simple formulation, the number of possible partitions can be enormous. If there are n regions and there are two choices (software or hardware) for each one, then there are 2^n possible partitions. A mere 32 regions yield over 4 billion possibilities. Finding the optimal partition of this simple form is known to be NP-hard in general. Many other factors contribute to making the problem even harder, as will be discussed.

This chapter discusses issues involved in partitioning an application among microprocessor and coprocessor components. It considers two application categories: *sequential programs*, where an application is a program written in a sequential programming language such as C, C++, or Java and where partitioning maps critical functions and/or loops to coprocessors; and *parallel programs*, where an application is a set of concurrently executing tasks and where partitioning maps some of those tasks to coprocessors.

While designers today do mostly manual partitioning, automating the process has been an area of active study since the early 1990s (e.g., [10, 15, 26]) and continues to be intensively researched and developed. For that reason, we will begin the chapter with a discussion of the trend toward automatic partitioning.

26.1 THE TREND TOWARD AUTOMATIC PARTITIONING

Traditionally, designers have manually partitioned applications between microprocessors and custom coprocessors. Manual partitioning was in part necessitated by radically different design flows for microprocessors versus coprocessors. A microprocessor design flow typically involved developing code

in programming languages such as C, C++, or Java. In sharp contrast, a coprocessor design flow may have involved developing cleverly parallelized and/or pipelined datapath circuits, control circuits to sequence data through the datapath, memory circuits to enable rapid data access by the datapath, and then mapping those circuits to a particular ASIC technology. Thus, manual partitioning was necessary because partitioning was done early in the design process, well before a machine-readable or executable description of an application's desired behavior existed. It resulted in specifications for both the software design and the hardware design teams, both of which might then have worked for many months developing their respective implementations.

However, the evolution of synthesis and FPGA technologies is leading toward automated partitioning because the starting point of FPGA design has been elevated to the same level as that for microprocessors, as shown in Figure 26.2.

Current technology enables coprocessors to be realized merely by downloading bits onto an FPGA. Downloading takes just seconds and eliminates the months-long and expensive design step of mapping circuits to an ASIC. Furthermore, synthesis tools have evolved to automatically design coprocessors from high-level descriptions in hardware description languages (HDLs), such as VHDL or Verilog, or even in languages traditionally used to program microprocessors, such as C, C++, or Java. Thus, designers may develop a single machine-readable high-level executable description of an application's desired behavior and then partition that description between microprocessor and coprocessor parts, in a process sometimes called hardware/software codesign. New

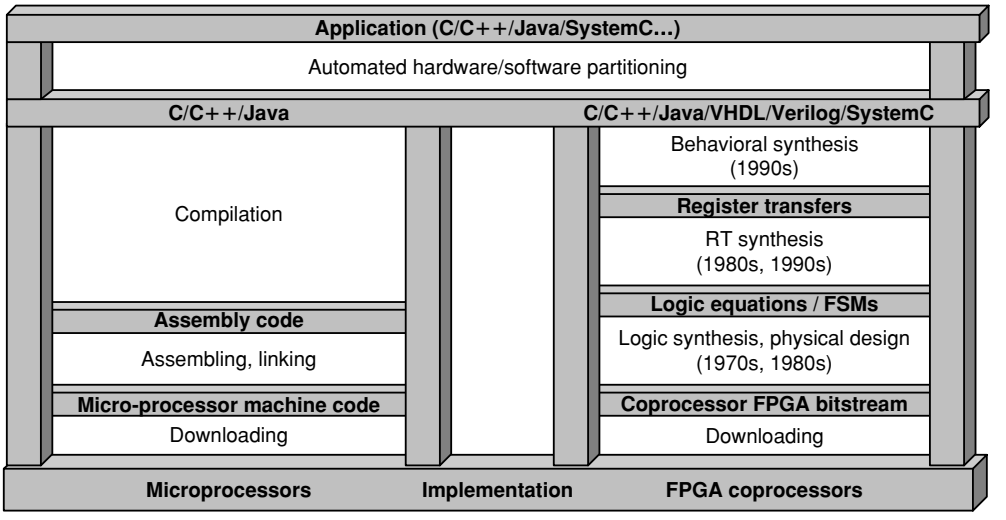

FIGURE 26.2 ■ The codesign ladder: evolution toward automated hardware/software partitioning due to synthesis tools and FPGA technologies enabling a similar design starting point, and similar implementation manner of downloading bits into a prefabricated device.

approaches, such as SystemC [14], which supports HDL concepts using C++, have evolved specifically to support it. With a single behavior description of an application, and automated tools to convert partitioned applications to coprocessors, automating partitioning is a logical next step in tool evolution. Some commercial automated hardware/software partitioning products are just beginning to appear [4, 7, 21, 27].

In the remainder of the chapter, many of the issues discussed relate to both manual and automatic partitioning, while some relate to automatic partitioning alone.

26.2 PARTITIONING OF SEQUENTIAL PROGRAMS

In a sequential program, the regions comprising an application's behavior are defined to execute sequentially rather than concurrently. For example, the semantics of the C programming language are such that its functions execute sequentially (though parallel execution is allowed as long as the results of the computation stay the same). Hardware/software partitioning of a sequential program involves speeding up certain regions by moving them to faster-executing FPGA coprocessors, yielding overall application speedup.

Hardware/software partitioning of sequential programs is governed to a large extent by the well-known Amdahl's Law [1] (described in 1967 by Gene Amdahl of IBM in the context of discussing the limits of parallel architectures for speeding up sequential programs). Informally, Amdahl's Law states that application speedup is limited by the part of the program *not* being parallelized. For example, if 75 percent of a program can be parallelized, the remaining nonparallelized 25 percent of the program limits the speedup to $100/25 = 4$ times speedup (usually written as $4x$) in the best possible case, even in the ideal situation of zero-time execution of the other 75 percent.

Amdahl's Law has been described more formally using the equation $\texttt{max\_speedup} = 1/(s + p/n)$, where p is the fraction of the program execution that can be parallelized; s is the fraction that remains sequential, $s + p = 1$; n is the number of parallel processors being used to speed up the parallelizable fraction; and `max_speedup` is the ideal speedup. In the 75 percent example, assuming that n is very large, we obtain $\texttt{max\_speedup} = 1/(0.25 + 0.75/n) = 1/(0.25 + \sim 0) = 4\text{x}$.

Amdahl's Law applies to hardware/software partitioning by providing speedup limits based on the regions *not* mapped to hardware. For example, if a region accounts for 25 percent of execution but is not mapped to hardware, then the maximum possible speedup obtainable by partitioning is 4x. Figure 26.3 illustrates that only when regions accounting for a large percentage of execution are mapped to hardware might partitioning yield substantial results. For example, to obtain 10x speedup, partitioning *must* map to hardware those regions accounting for *at least* 90 percent of an application's execution time.

Fortunately, most of the execution time for many applications comes from just a few regions. For example, Figure 26.4 shows the average execution time

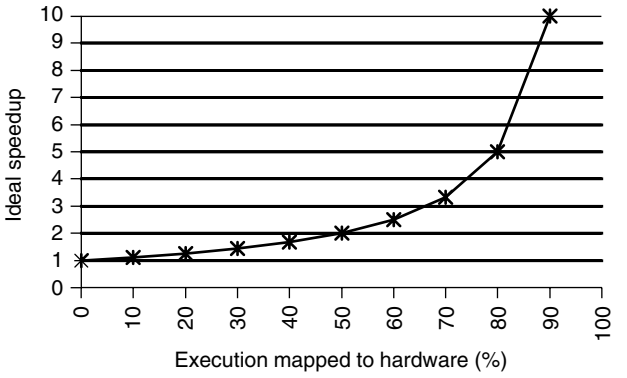

FIGURE 26.3 ■ Hardware/software partitioning speedup following Amdahl's Law.

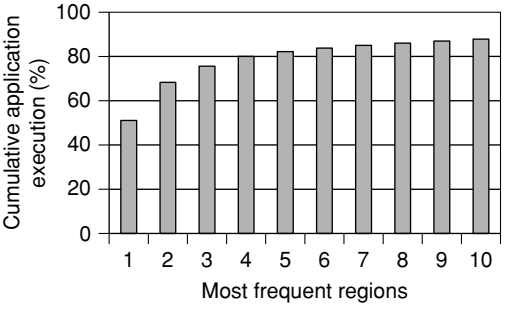

FIGURE 26.4 ■ Ideal speedups achievable by moving regions (loops) to hardware, averaged for a variety of embedded system benchmark suites (MediaBench, Powerstone, and Netbench).

contribution for the first n regions (in this case loops) for several dozen standard embedded system application benchmarks, all sequential programs. Note that the first few regions account for 75 to 80 percent of the execution time. The regions are roughly equal in size following the well-known informal "90–10" rule, which states that that 90 percent of a program's execution time is spent in 10 percent of its code. Thus, hardware/software partitioning of sequential programs generally must sort regions by their execution percentage and then consider moving the highest contributing regions to hardware.

A corollary to Amdahl's Law is that if a region is moved to hardware, its actual speedup limits the remaining possible speedup. For example, consider a region accounting for 80 percent of execution time that, when moved to hardware, runs only 2x faster than in software. Such a situation is equivalent to 40 percent of the region being sped up ideally and the other 40 percent not being sped up at all. With 40 percent not sped up, the ideal speedup obtainable by partitioning of the remaining regions (the other 20 percent) is limited

to a mere 100 percent/40 percent = 2.5x. For this reason, hardware/software partitioning of sequential programs generally must focus on obtaining very large speedups of the highest-contributing regions.

Amdahl's Law therefore greatly prunes the solution space that partitioning of sequential programs must consider—good solutions must move the biggest-contributing regions to hardware and greatly speed them up to yield good overall application speedups.

Even with this relatively simple view, several issues make the problem of hardware/software partitioning of sequential programs quite challenging. Those issues, illustrated in Figure 26.5(a–e), include determining critical region

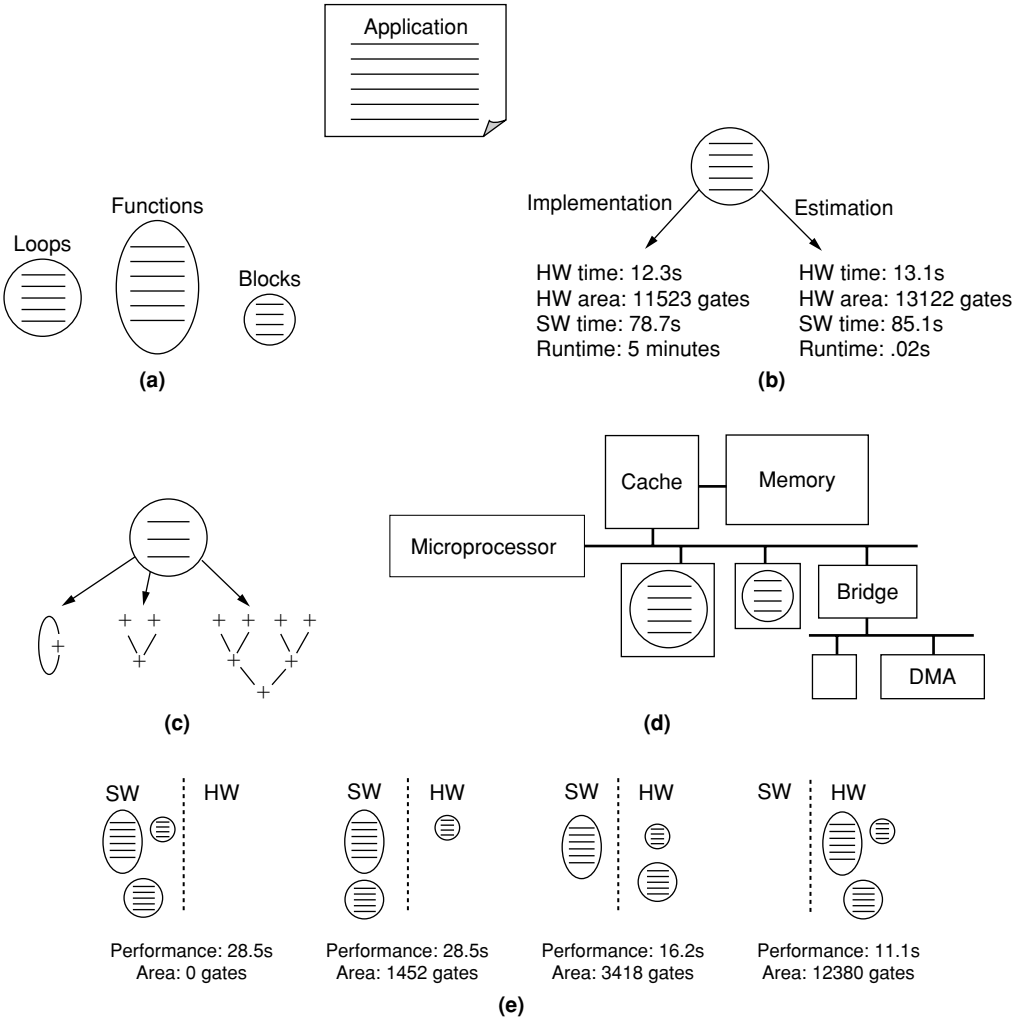

FIGURE 26.5 ■ Hardware/software partitioning: (a) granularity; (b) partition evaluation; (c) alternative region implementations; (d) implementation models; (e) exploration.

granularity (a), evaluating partitions (b), considering multiple alternative implementations of a region (c), determining implementation models (d), and exploring the partitioning solution space (e).

26.2.1 Granularity

Partitioning moves some code regions from a microprocessor to coprocessors. A first issue in defining a partitioning approach is thus to determine the granularity of the regions to be considered. *Granularity* is a measure of the amount of functionality encapsulated by a region, which is illustrated in Figure 26.6.

A key trade-off involves coarse versus fine region granularity [11]. Coarser granularity simplifies partitioning by reducing the number of possible partitions, enables more accurate estimates during partitioning by considering more computations when creating those estimates (and thus reducing inaccuracy when combining multiple estimates for different regions into one), and reduces inter-region communication. On the other hand, finer granularity may expose better

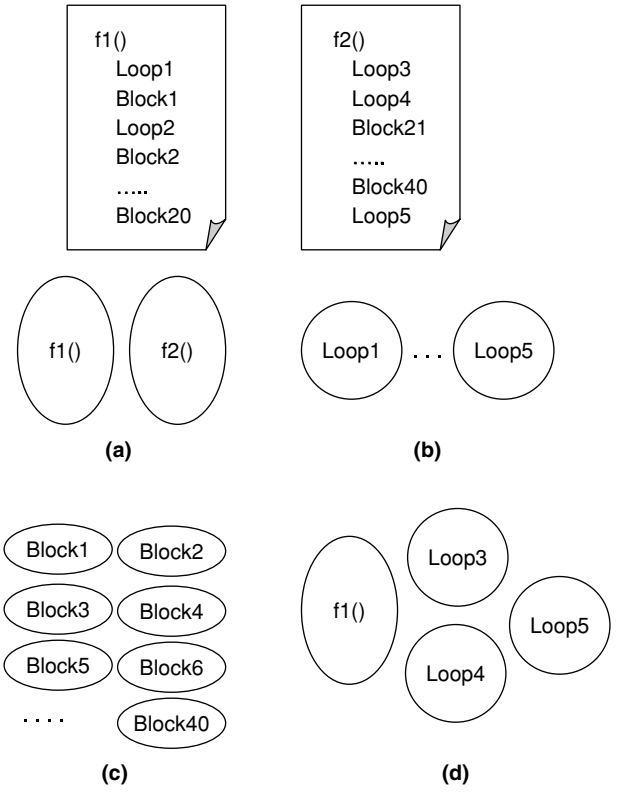

FIGURE 26.6 ■ The region granularities of an application (*top*): (a) functions; (b) loops; (c) blocks; (d) heterogeneous combination. Finer granularities may expose better solutions, at the expense of a more complex partitioning problem and more difficult estimation challenges.

partitions that would not otherwise be possible. Early automated partitioning research considered fine granularities of arithmetic operations or statements, while more recent work typically considers coarser granularities involving basic blocks, loops, or entire functions.

Coarse granularity simplifies the partitioning problem by reducing the number of possible partitions. Take, for example, an application with two 1000-line C functions, like the one shown in Figure 26.6 (*top*), and consider partitioning at the granularity of functions, loops, or basic blocks. The granularity of functions involves only two regions, as shown in Figure 26.6(a), and the granularity of loops involves five regions, as shown Figure 26.6(b). However, the granularity of the basic block may involve many tens or hundreds of regions, as shown in Figure 26.6(c). If partitioning simply chooses between hardware and software, then two regions would yield 2*2 = 4 possible partitions, while just 32 regions would involve 2*2*2*…*2 (32 times) possible partitions, or over four billion.

Coarse granularity also enables more accurate early estimations of a region's performance, size, power, and so forth. For example, an approach using function granularity could individually presynthesize the two previously mentioned functions to FPGAs before partitioning, gathering performance and size data. During partitioning, it could simply estimate that, for the case of partitioning both functions to the FPGA, the two functions' performances would stay the same and their sizes would add. This estimate is not entirely accurate because synthesizing both functions could involve interactions between the function's implementations that would impact performance and size, but it is likely *reasonably* accurate. In contrast, similar presynthesis and performance/size estimates for basic blocks would yield grossly inaccurate values because multiple basic blocks would actually be synthesized into a combined circuit having extensive sharing among the blocks, bearing little resemblance to the individual circuits presynthesized for each block.

However, finer granularity may expose better partitions that otherwise would not be possible. In the two-function example just described, perhaps the best partition would move only half of one function to hardware—an option not possible at the coarse granularity of functions but possible at finer granularities of loops or basic blocks.

Manual partitioning often involves initially considering a "natural" granularity for an application. An application may consist of dozens of functions, but a designer may naturally categorize them into just a few key high-level functions. A data-processing application, for example, may naturally consist of several key high-level functions: acquire, decompress, transform, compress, and transmit. The designer may first try to partition at that natural granularity before considering finer granularities.

Granularity may be restricted to one region type, but can instead be *heterogeneous*, as shown in Figure 26.6(d). For example, in the previous two-function example from Figure 26.6 (*top*), one function may be treated as a region while the other may be broken down so that its loops are each considered as a region. A particular loop may even be broken down so that its basic blocks are individually considered as regions. Thus, for a single application, regions considered

for movement to hardware may include functions, loops, and basic blocks. With heterogeneous granularity, preanalysis of the code may select regions based on execution time and size, breaking down a region with very high execution time or large size.

Furthermore, while granularity can be predetermined statically, it can also be determined *dynamically* during partitioning [16]. Thus, an approach might start with coarse-grained regions and then decompose specific regions deemed to be critical during partitioning.

Granularity need not be restricted to regions defined by the language constructs such as functions or loops, used in the original application description. Transformations, some being well-known compiler transformations, may be applied to significantly change the original description. They include function inlining (replacing a function call with that function's statements), function "exlining" (replacing statements with a function call), function cloning (making multiple copies of a function for use in different places), function specialization (creating versions of a function with constant parameters), loop unrolling (expanding a loop's body to incorporate multiple iterations), loop fusion (merging two loops into one), loop splitting (splitting one loop into two), code hoisting and sinking (moving code out of and into loops), and so on.

26.2.2 Partition Evaluation

The process of finding a good partition is typically iterative, involving consideration and evaluation of certain partitions and then decisions as to which partitions to consider next. *Evaluation* determines a partition's design metric values. A *design metric* is a measure of a partition. Common metrics include performance, size, and power/energy. Other metrics include implementation cost, engineering cost, reliability, maintainability, and so on.

Some design metrics may need to be *optimized*, meaning that partitioning should seek the best possible value of a metric. Other design metrics may be *constrained*, meaning that partitioning must meet some threshold value for a metric. An *objective function* is one that combines multiple metric values into a single number, known as *cost*, which the partitioning may seek to minimize. A partitioning approach must define the metrics and constraints that can be considered, and define or allow a user to define an objective function.

Evaluation can be a complex problem because it must consider several implementation factors in order to obtain accurate design metric values. Among others, these factors include determining the communication time between regions that transfer data (thus requiring knowledge of the communication structure), considering clock cycle lengthening caused by multiple application regions sharing hardware resources (which may introduce multiplexers or longer wires), and the like.

The key trade-off in evaluation involves estimation versus implementation. Estimating design metric values is faster and so enables consideration of more possible partitions. Obtaining the values through implementation is more accurate and thus ensures that partitioning decisions are based on sound evaluations.

Estimation involves some characterization of an application's regions before partitioning and then, during partitioning, quickly combining the characterizations into design metric values. The previous section on granularity discussed how two C function regions could be characterized for hardware by synthesizing each region individually to an FPGA, resulting in a characterization of each region consisting of performance and size data. Then a partition with multiple regions in hardware could be evaluated simply by assuming that each region's performance is the same as the predetermined performance and by adding any hardware-mapped region sizes together to obtain total hardware size. Estimation for software can be done similarly, using compilation rather than synthesis for characterization.

Nevertheless, while estimation typically works well for software [24], the nature of hardware may introduce significant inaccuracy into an estimation approach because multiple regions may actually share hardware resources, thus intertwining their performance and size values [9,18]. Alternatively, implementation as a means of evaluation involves synthesizing actual hardware circuits for a given partition's hardware regions. Such synthesis thus accounts for hardware sharing and other interdependencies among the regions. However, synthesis is time consuming, requiring perhaps tens of seconds, minutes, or even hours, restricting the number of partitions that can be evaluated.

Many approaches exist between the two extremes just described. Estimation can be improved with more extensive characterization, incorporating much more detail than just performance and size. Characterization may, for example, describe what hardware resources a region utilizes, such as two multipliers or 2 Kbytes of RAM. Then estimation can use more complex algorithms to combine region characterizations into actual design metric values, such as that the regions may share resources such as multipliers (possibly introducing multiplexers to carry out such sharing) or RAM. These algorithms yield higher accuracy but are still much faster than synthesis. Alternatively, synthesis approaches can be improved by performing a "rough" rather than a complete synthesis, using faster heuristics rather than slower, but higher-optimizing heuristics, for example.

Evaluation need not be done in a single exploration loop of partitioning, but can be *heterogeneous*. An outer exploration loop may be added to partitioning that is traversed less frequently, with the inner exploration loop considering thousands of partitions (if automated) and using estimation for evaluation, while the outer exploration loop considers only tens of partitions that are evaluated more extensively using synthesis. The inner/outer loop concept can of course be extended to even more loops, with the inner loops examining more partitions evaluated quickly and the outer loops performing increasingly in-depth synthesis on fewer partitions.

Furthermore, evaluation methods can change *dynamically* during partitioning. Early stages in the partitioning process may use fast estimation techniques to map out the solution space and narrow in on particular sections of it, while later stages may utilize more accurate synthesis techniques to fine-tune the solution.

26.2.3 Alternative Region Implementations

Further adding to the partitioning challenge is the fact that a given region may have *alternative region implementations* in hardware rather than just one implementation, as assumed in the previous sections. For example, Figure 26.7 (*top*) shows a particular function that performs 100 multiplications. A fast but large hardware implementation may use 100 multipliers, as shown in Figure 26.7(a). The much smaller but much slower hardware implementation in Figure 26.7(b) uses only 1 multiplier. Numerous implementation alternatives exist between those two extremes, such as having 2 multipliers as in Figure 26.7(c), 10 multipliers, and so on. Furthermore, the function may be implemented in a pipelined or non-pipelined manner. Utilized components may be fast and large (e.g., array-style multipliers or carry-lookahead adders) or small and slow (e.g., shift-and-add multipliers or carry-ripple adders). Many other alternatives exist.

A key trade-off involves deciding how many alternative implementations to consider during partitioning. More alternatives greatly expand the number of possible partitions and thus may possibly lead to improved results. However, they also expand the solution space tremendously. For example, 8 regions each with one hardware implementation yield $2^8 = 256$ possible partitions. If each

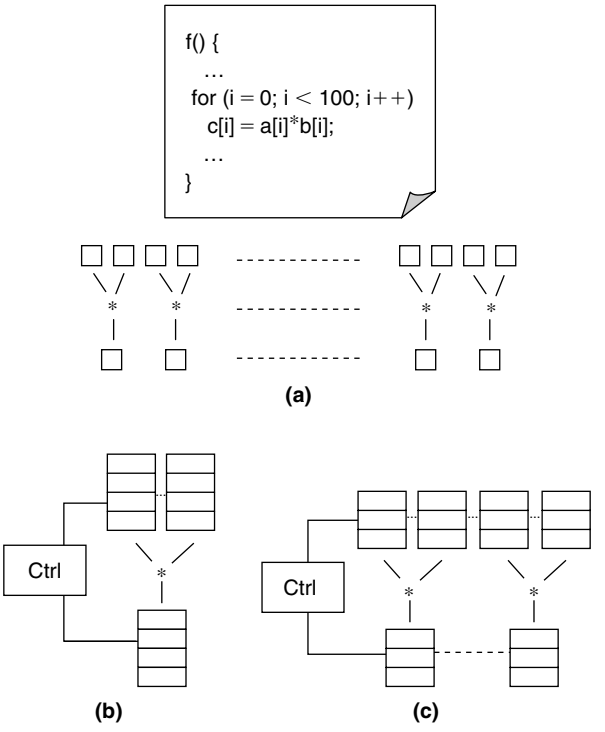

FIGURE 26.7 ■ Alternative region implementations for an original application (*top*) requiring 100 multiplications: (a) 100 multipliers; (b) 1 multiplier; (c) 2 multipliers. Alternative region implementations may have hugely different performances and sizes.

region instead has 4 possible hardware implementations, then it has 5 possible implementations (1 software and 4 hardware implementations), yielding 5^8, or more than 300,000, possible partitions.

Most automated hardware/software partitioning approaches consider one possible hardware implementation per region. Even then, a question exists as to which one to consider for that region: the fastest, the smallest, or some alternative in the middle? Some approaches do consider multiple alternative implementations, perhaps selecting a small number that span the possible space, such as small, medium, and large [5].

As we saw with granularity and evaluation, the number of alternative implementations considered can also be *heterogeneous*. Partitioning may consider only one alternative for particular regions and multiple alternatives for other regions deemed more critical.

Furthermore, as we saw with granularity and evaluation, the number of alternative implementations can change *dynamically* as well. Partitioning may start by considering only a few alternatives per region and then consider more for particular regions as partitioning narrows in on a solution.

Sometimes obtaining alternative implementations of an application region may require the designer to write several versions of it, each leading to one or more alternatives. In fact, a designer may have to write different region versions for software and hardware because a version that executes fast in software may execute slow in hardware, and vice versa. That difference is due to software's fundamental sequential execution model that demands clever sequential algorithms, while hardware's inherently parallel model demands parallelizable algorithms.

26.2.4 Implementation Models

Partitioning moves critical microprocessor software regions to hardware coprocessors. Different *implementation models* define how the coprocessors are integrated with the microprocessor and with one another [6], enlarging the possible solution space for partitioning and greatly impacting performance and size.

One implementation model parameter is whether coprocessor execution and microprocessor execution overlap or are mutually exclusive. In the overlapping model, the microprocessor activates a coprocessor and may then continue to execute concurrently with it (if the data dependencies of the application allow). In the mutually exclusive model, the microprocessor waits idly until the coprocessor finishes, at which time the microprocessor resumes execution.

Figure 26.8(a) illustrates the execution of both models. Overlapping may improve overall performance, but mutual exclusivity simplifies implementation by eliminating issues related to memory contention, cache coherency, and synchronization—the coprocessor may even access cache directly. In many partitioned implementations, the coprocessor executes for only a small fraction of the total application cycles, meaning that overlapping gains little performance improvement. When the microprocessor and coprocessor cycles are closer to

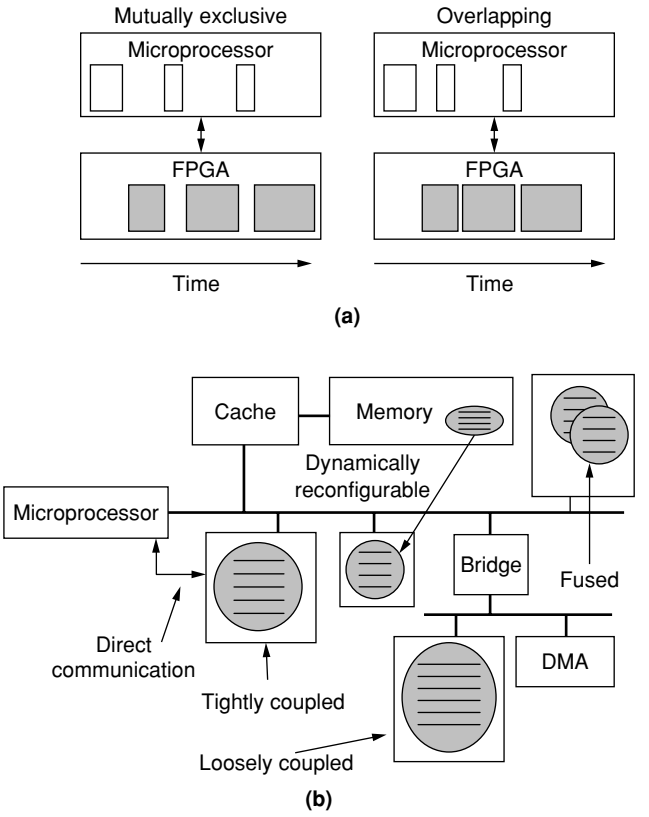

FIGURE 26.8 ■ Implementation models: (a) mutually exclusive and overlapping. (b) implementation model parameters.

being equal, overlapping may improve performance, up to a limit of 2 times, of course. Similarly, the execution of coprocessors relative to one another may be overlapped or mutually exclusive.

A second implementation model parameter involves communication methods. The microprocessor and coprocessors may communicate through memory and share the same data cache, or the microprocessor may communicate directly with the FPGA through memory-mapped registers, queues, fast serial links, or some combination of those mechanisms.

Another implementation model parameter is whether multiple coprocessors are implemented separately or are fused. In a separate coprocessor model, each critical region is synthesized to its own controller and datapath. In a fused model, the critical regions are synthesized into a single controller and datapath. The fused model may reduce size because the hardware resources are shared, but it may result in performance overhead because of a longer critical path as well as the need to run at the slowest clock frequency of all the regions.

Certain coprocessors can be fused and others left separate. Furthermore, fusing need not be complete—two coprocessors can share key components, such as a floating-point unit, but otherwise be implemented separately.

Yet another model parameter is whether coprocessors and the microprocessor are tightly or loosely coupled. Tightly coupled coprocessors may coexist on the microprocessor memory bus or may even have direct access to microprocessor registers. Loosely coupled, they may access microprocessor memory through a bridge, adding several cycles to data accesses. Both couplings can coexist in a single implementation.

FPGAs add a particularly interesting model parameter to partitioning—dynamic reconfiguration—which replaces an FPGA circuit with another circuit during runtime by swapping in a new FPGA configuration bitstream [2]. In this way, not all of an application's coprocessors need to simultaneously coexist in the FPGA. Instead, one subset of the application's required coprocessors may initially be loaded into the FPGA, but, as the application continues to execute, that subset may be replaced by another subset needed later in the application's execution. Reconfiguration increases the effective size of an FPGA, thus enabling better performance when more application regions are partitioned to it or, alternatively, enabling use of a smaller and hence cheaper FPGA with a runtime overhead required to swap in new bitstreams. In some cases, this overhead may limit the benefits of reconfiguration and should therefore be considered during partition evaluation.

Figure 26.8(b) illustrates some of the different implementation model parameters, including communication methods, fused regions, and tightly/loosely coupled coprocessors. Often these parameters are fixed prior to partitioning, but can also be explored dynamically during partitioning to determine the best implementation model for a given application and given constraints.

26.2.5 Exploration

Exploration is the searching of the partition solution space for a good partition. As mentioned before, it is at present mostly a manual task, but automated techniques are beginning to mature. This section discusses automated exploration techniques for various formulations of the partitioning problem.

Simple formulation

A simple and common form of the hardware/software partitioning problem consists of n regions, each having a software runtime value, a hardware runtime value, and a hardware size. It assumes that all values are independent of one another (so if two regions are mapped to hardware, their hardware runtime and size values are unchanged); it assumes that communication times are constant regardless of whether a region is implemented as software or hardware (such as when all regions use the same interface to a shared memory); and it seeks to minimize total application runtime subject to a hardware size constraint (assuming no dynamic reconfiguration).

Although this problem is known to be NP-hard, it can be solved by first mapping it to the well-known *0-1 knapsack problem* [20]. The 0-1 knapsack problem involves a knapsack with a specified weight capacity and a set of items, each with a weight and a profit. The goal is to select which items to place in the knapsack such that the total profit is maximized without violating the weight capacity. For hardware/software partitioning, regions correspond to items, the FPGA size constraint corresponds to the knapsack capacity, an implementation's size corresponds to an item's weight, and the speedup obtained by implementing a region in hardware instead of software corresponds to an item's profit.

Thus, algorithms that solve the 0-1 knapsack problem solve the simple form of the hardware/software partitioning problem. The 0-1 knapsack problem is NP-hard, but efficient optimal algorithms exist for relatively large problem sizes. One of these is a well-known dynamic programming algorithm [12] having runtime complexity of $O(A^*n)$, where A is the capacity and n is the number of items. Alternatively, integer linear programming (ILP) [22] may be used. ILP solvers perform extensive solution space pruning to reduce exploration time.

For problems too big for either such optimal technique, heuristics may be utilized. A *heuristic* finds a good, but not necessarily the optimal, solution, while an *algorithm* finds the optimal solution. A common heuristic for the 0-1 knapsack problem is a greedy one. A greedy heuristic starts with an initial solution and then makes changes only if they seem to improve the solution. It sorts each item based on the ratio of profit to weight and then traverses the sorted list, placing an item in the knapsack if it fits and skipping it otherwise, terminating when reaching the knapsack capacity or when all items have been considered. This heuristic has $O(nlgn)$ time complexity, allowing for fast automated partitioning of thousands of regions or feasible manual partitioning of tens of regions. Furthermore, the heuristic has been shown to commonly obtain near-optimal results in the situation when a few items have a high profit to weight ratio. In hardware/software partitioning terms, that situation corresponds to the existence of regions that are responsible for the majority of execution time and require little hardware area, which is often the case.

Formulation with asymmetric communication and greedy/nongreedy automated heuristics

A slightly more complex form of the hardware/software partitioning problem considers cases where communication times between regions change depending on the partitioning, with different required times for communication depending on whether the regions are both in software or both in hardware, or are separated, with one in software and one in hardware. This form of the problem can be mapped to the well-known graph bipartitioning problem.

Graph bipartitioning divides a graph into two sets in order to minimize an objective function. Each graph node has two weights, one for each set. Edges may have three different weights: two weights associated with nodes connected in the same set (one weight for each set) and one for nodes connected between sets. Typically, the objective function is to minimize the sum of all node and edge

weights using the appropriate weights for a given partition. Graph bipartitioning is NP-hard.

ILP approaches may be used for automatically obtaining optimal solutions to the graph bipartitioning problem. Heuristics may be used when ILP is too time consuming. A simple greedy heuristic for graph bipartitioning starts with some initial partition, perhaps random or all software. It then determines the cost improvement of moving each node from its present set to the opposite set and then moves the node yielding the best improvement. The heuristic repeats these steps until no move yielding an improvement is found. Given n nodes, a basic form of such a heuristic has $O(n^2)$ runtime complexity. Techniques to update the existing cost improvement values can reduce the complexity to $O(n)$ in practice [25].

More advanced heuristics seek to overcome what are known as "local minima," accepting solution-worsening moves in the hope that they will eventually lead to an even better solution. For example, Figure 26.9 illustrates a heuristic that accepts some solution-worsening changes to escape a local minimum and eventually reach a better solution. A common situation causing a local minimum involves two items such that moving only one item worsens the solution but moving both improves it.

A well-known category of nongreedy heuristic used in partitioning is known as *group migration* [11], which evolved from an initial heuristic by Kernighan–Lin. Like the previous greedy heuristic, group migration starts with an initial partition and determines the cost improvement of moving each node from its present set to the opposite set. The group migration heuristic then moves the node yielding the best improvement (like the greedy heuristic) or yielding the *least worsening* (including zero cost change) if no improving move exists. Accepting such worsening moves enables local minima to be overcome. Of course, such a heuristic would never terminate, so group migration ensures termination by locking a node after it is moved. Group migration moves each node exactly once in what is referred to as an iteration, and an iteration has complexity of $O(n^2)$ (or $O(n)$ if clever techniques are used to update cost improvements after each

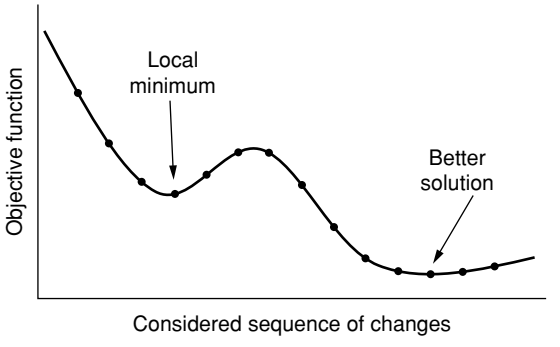

FIGURE 26.9 ■ Solution-worsening moves accepted by a nongreedy heuristic to escape local minima and find better solutions.

move). If an iteration ultimately leads to an improvement, then group migration runs another iteration. In practice, only a few iterations, typically less than five, can be run before no further improvement can be found.

The previous discussions of heuristics ignore the time required by partition evaluation. The heuristics therefore may have even higher runtime complexity unless care is taken to incorporate fast incremental evaluation updates during exploration.

Complex formulations and powerful automated heuristics

Increasingly complex forms of the hardware/software partitioning problem integrate more parameters related to the earlier mentioned issues of exploration—granularity, evaluation, alternative region implementation, and implementation models. For example, the earlier mentioned dynamic granularity modifications, such as decomposing a given region into smaller regions, or even applying transformations to an application such as function inlining, can be applied during partitioning. The partitioning problem can consider different couplings of coprocessors, may also consider coprocessor fusing, and can support dynamic reconfiguration. When one considers the multitude of possible parameters that can be integrated with partitioning, the size of the solution space is mind-boggling. Searching that space for the best solution becomes a tremendous combinatorial optimization challenge, likely requiring long-running search heuristics.

At this point, it may be interesting to note that hardware/software partitioning brings together two previously separate research fields: compilers and CAD (computer-aided design). Compilation techniques tend to emphasize a quick series of *transformations* applied to an application's description. In contrast, CAD techniques tend to emphasize a long-running iterative *search* of enormous solution spaces. One possible reason for these different perspectives is that compilers were generally expected to run quickly, in seconds or at most minutes, because they were part of a design loop in which compilation was applied perhaps dozens or hundreds of times a day as programs were developed. In contrast, CAD optimization techniques were part of a much longer design loop. Running CAD optimization tools for hours or even days was perfectly acceptable because that time was still small compared to the weeks or months required to manufacture chips. Furthermore, the very nature of coprocessor design meant that a designer was extremely interested in high performance, so longer tool runtimes were acceptable if they optimized an implementation.

Hardware/software partitioning merges compilation and synthesis into a single framework. In some cases, compiler-like runtimes of seconds must be achieved. In other cases, CAD-like runtimes of hours may be acceptable. Approaches to partitioning may span that range. Highly complex partitioning formulations will likely require moving away from the fast linear time algorithms and heuristics described earlier and toward longer-running powerful search heuristics.

A popular powerful and general search heuristic is *simulated annealing* [17]. The simulated annealing heuristic starts with a random solution and then randomly makes some change to it, perhaps moving a region between software

and hardware, choosing an alternative implementation for a particular region, decomposing a particular region into finer-grained regions, performing a transformation on the original regions, and so forth, and evaluates the cost (as determined by an objective function) of the new partition obtained from that change. If the change improves the cost, it is accepted (i.e., the change is made). If the change worsens the cost, the seemingly "bad" change is accepted with some probability. The key feature of simulated annealing is that the probability of accepting a seemingly bad move decreases as the approach proceeds, with the pattern of decrease determined by some parameters provided to the annealing process that eventually causes it to narrow in on a good solution. Simulated annealing typically must evaluate many thousands or millions of solutions in order to arrive at a good one and thus requires very fast evaluation methods.

The complexity of simulated annealing is generally dependent on the problem instance. With properly set parameters, it can achieve near-optimal solutions on very large problems in long but acceptable runtimes. Faster machines have made simulated annealing an increasingly acceptable search heuristic for a wider variety of problems—it can complete in just seconds for many problem instances.

The simulated annealing heuristic is known as a neighborhood search heuristic because it makes local changes to an existing solution. Tabu search [13] is an effective method for improving neighborhood search. Meaning "forbidden," Tabu maintains a list of recently seen, Tabu, solutions. When considering a change to an existing solution, it disregards any change that would yield a solution on the Tabu list. This prevents cycling among the same solutions and has been shown to yield improved results in less time. The Tabu list concept can also be applied on a broader scale, maintaining a long-term history of considered solutions in order to increase solution diversity. Tabu search can improve neighborhood search heuristic runtimes during hardware/software partitioning by a factor of 20x [8].

Other issues

Because implementing an application as software generally requires a smaller size and less designer effort, most approaches to exploration start with an all-software implementation and then explore the mapping of critical application regions to hardware. However, in some cases, such as when the application is written specifically for hardware, an approach may start with an all-hardware implementation and then move noncritical application regions to software to reduce hardware size.

Furthermore, when an application is originally written for software implementation, some of its regions may not be suitable for hardware implementation. For example, application regions that utilize recursive function calls, pointer-based data structures, or dynamic memory allocation may not be easy to implement as a hardware circuit. Some research efforts are beginning to address these problems by developing new synthesis techniques that support a wider range of program constructs and behavior. Alternatively, designers sometimes

write (or rewrite) critical regions such that those regions are well suited for circuit implementation.

26.3 PARTITIONING OF PARALLEL PROGRAMS

In parallel programs, the regions that make up an application are defined to execute concurrently, as opposed to sequentially. Such regions are often called *tasks* or *processes*. For some applications, expressing behavior using tasks may result in a more parallel implementation and hence in faster application performance. For example, an MPEG2 decoder may be described as several tasks, such as motion compensation, dequantization, or inverse discrete cosine transform, that can be implemented in a pipelined manner.

Numerous parallel programming models have been considered for hardware/software partitioning, among others, synchronous dataflow, dynamic dataflow, Kahn process networks, and communicating sequential processes.

26.3.1 Differences among Parallel Programming Models

While hardware/software partitioning of parallel programs has many similarities to partitioning for sequential programs, several key differences exist.

Granularity
Partitioning of parallel programs typically treats each task as a region, meaning that the granularity is quite coarse. In some cases, decomposing a task into finer granularity may be considered.

Evaluation
Parallel programs often involve multiple performance constraints, with particular tasks or sets of tasks having unique performance constraints of their own. Furthermore, estimations of performance must consider the scheduling of tasks on processors, which is not an issue for sequential programs because regions in these programs are not concurrent.

Alternative region implementations
Given the coarse granularity of tasks, considering alternative implementations becomes even more important, as the variations among the alternatives can be huge.

Implementation models
Because tasks are inherently concurrent, partitioning of parallel programs typically uses parallel execution models in their implementations, meaning that microprocessors and coprocessors run concurrently rather than mutually exclusively and meaning that coprocessors may be arranged to form high-level pipelines. Partitioning of parallel programs is less likely to consider fusing multiple coprocessors into one because fusing eliminates concurrency.

Parallel program partitioning introduces a new aspect to exploration—scheduling. When mapping multiple tasks to a single microprocessor, partitioning must carry out the additional step of scheduling to determine when each task will execute. Scheduling tasks to meet performance constraints is known as *real-time scheduling* and is a heavily studied problem [3].

Including partitioning during scheduling results in a more complex problem. Such partitioning often considers more than just one microprocessor as well and even different types of microprocessors. It may even consider different numbers and types of memories and different bus structures connecting memories to processors.

Parallel partitioning must also pay more attention to the data storage requirements between processors. Queues may be introduced between processors, the sizes of those queues must be determined, and their implementation (e.g., in shared memory or in separate hardware components) must be decided.

Exploration

More complex issues in the hardware/software partitioning problem—such as scheduling, different granularities, different evaluation methods, alternative region implementations, and different numbers and connections of microprocessors/memories/buses—require more complex solution approaches. Most modern automatic partitioning research considers one or a few extensions to basic hardware/software partitioning and develops custom heuristics to solve the new formulations in fast compiler-like runtimes. However, as more complex forms of partitioning are considered, more powerful search heuristics with longer runtimes, such as simulated annealing or search algorithms tuned to the problem formulation, may be necessary.

26.4 SUMMARY AND DIRECTIONS

Developing an approach for hardware/software partitioning requires the consideration of granularity, evaluation, alternative region implementations, implementation models, exploration, and so forth, and each such issue involves numerous options. The result is a tremendously large partition solution space and a huge variety of approaches to finding good partitions. While much research into automated hardware/software partitioning has occurred over the past decades, most of the problem's more complex formulations have yet to be considered. A key future challenge will be the development of effective partitioning approaches for these increasingly complex formulations.

As FPGAs continue to enter mainstream embedded, desktop, and server computing, incorporating automated hardware/software partitioning into standard software design flows becomes increasingly important. One approach to minimizing the disruption of standard software design flows is to incorporate partitioning as a backend tool that operates on a final binary, allowing continued use of existing programming languages and compilers and supporting the use

of assembly and even object code. Such binary-level partitioning [23] requires powerful decompilation methods to recover high-level regions such as functions and loops. Binary-level partitioning even opens the door for dynamic partitioning, wherein on-chip tools transparently move software regions to FPGA coprocessors, making use of new lean, just-in-time compilers for FPGAs [19].

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967.

[2] J. Burns, A. Donlin, J. Hogg, S. Singh, M. De Wit. A dynamic reconfiguration runtime system. *Proceedings of the Symposium on FPGA-Based Custom Computing Machines*, 1997.

[3] G. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic, 1997.

[4] S. Chappell, C. Sullivan. Handel-C for co-processing and co-design of field programmable system on chip. *Proceedings of Workshop on Reconfigurable Computing and Applications*, 2002.

[5] K. Chatha, R. Vemuri. An iterative algorithm for partitioning, hardware design space exploration and scheduling of hardware-software systems. *Design Automation for Embedded Systems* 5(3–4), 2000.

[6] K. Compton, S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys* 34(2), 2002.

[7] CriticalBlue. *http://www.criticalblue.com*.

[8] P. Eles, Z. Peng, K. Kuchchinski, A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems* 2(1), 1997.

[9] R. Enzler, T. Jeger, D. Cottet, G. Tröster. High-level area and performance estimation of hardware building blocks on FPGAs. *Lecture Notes in Computer Science* 1896, 2000.

[10] R. Ernst, J. Henkel. Hardware-software codesign of embedded controllers based on hardware extraction. *Proceedings of the International Workshop on Hardware/Software Codesign*, 1992.

[11] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems,* Prentice-Hall, 1994.

[12] P. C Gilmore, R. E Gomory. The theory and computation of knapsack functions. *Operations Research* 14, 1966.

[13] F. Glover. Tabu search, part I. *Operations Research Society of America Journal on Computing* 1, 1989.

[14] T. Grotker, S. Liao, G. Martin, S. Swan. *System Design with System C*. Springer-Verlag, 2002.

[15] R. Gupta, G. De Micheli. System-level synthesis using re-programmable components. *Proceedings of the European Design Automation Conference*, 1992.

[16] J. Henkel, R. Ernst. A hardware/software partitioner using a dynamically determined granularity. *Design Automation Conference*, 1997.

[17] S. Kirkpatrick, C. Gelatt, M. Vecchi. Optimization by simulated annealing. *Science* 220(4598), May 1983.

[18] Y. Li, J. Henkel. A framework for estimation and minimizing energy dissipation of embedded HW/SW systems. *Design Automation Conference*, 1998.
[19] R. Lysecky, G. Stitt, F. Vahid. Warp processors. *Transactions on Design Automation of Electronic Systems* 11(3), 2006.
[20] S. Martello, P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, 1990.
[21] Poseidon Design Systems, Inc. *http://www.poseidon-systems.com/index.htm*.
[22] A. Schrijver. *Theory of Linear and Integer Programming*, Wiley, 1998.
[23] G. Stitt, F. Vahid. New decompilation techniques for binary-level co-processor generation. *Proceedings of the International Conference on Computer-Aided Design*, 2005.
[24] K. Suzuki, A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. *Design Automation Conference*, 1996.
[25] F. Vahid, D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on VLSI Systems* 3(3), 1995.
[26] F. Vahid, D. Gajski. Specification partitioning for system design. *Design Automation Conference*, 1992.
[27] XPRES Compiler. *http://www.tensilica.com/products/xpres.htm*.