Binary Synthesis

GREG STITT and FRANK VAHID
University of California, Riverside

Recent high-level synthesis approaches and C-based hardware description languages attempt to improve the hardware design process by allowing developers to capture desired hardware functionality in a well-known high-level source language. However, these approaches have yet to achieve wide commercial success due in part to the difficulty of incorporating such approaches into software tool flows. The requirement of using a specific language, compiler, or development environment may cause many software developers to resist such approaches due to the difficulty and possible instability of changing well-established robust tool flows. Thus, in the past several years, synthesis from binaries has been introduced, both in research and in commercial tools, as a means of better integrating with tool flows by supporting all high-level languages and software compilers. Binary synthesis can be more easily integrated into a software development tool-flow by only requiring an additional backend tool, and it even enables completely transparent dynamic translation of executing binaries to configurable hardware circuits. In this article, we survey the key technologies underlying the important emerging field of binary synthesis. We compare binary synthesis to several related areas of research, and we then describe the key technologies required for effective binary synthesis: decompilation techniques necessary for binary synthesis to achieve results competitive with source-level synthesis, hardware/software partitioning methods necessary to find critical binary regions suitable for synthesis, synthesis methods for converting regions to custom circuits, and binary update methods that enable replacement of critical binary regions by circuits.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; C.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design*

General Terms: Design, Performance

Additional Key Words and Phrases: Binary synthesis, synthesis from software binaries, hardware/software partitioning, hardware/software codesign, FPGA, configurable logic, warp processors

ACM Reference Format:

Stitt, G. and Vahid, F. 2007. Binary synthesis. ACM Trans. Des. Autom. Electron. Syst. 12, 3, Article 34 (August 2007), 30 pages. DOI = 10.1145/1255456.1255471 http://doi.acm.org/10.1145/1255456.1255471

F. Vahid is also affiliated with the Center for Embedded Computer Systems at the University of California, Irvine.

This research was supported by the National Science Foundation (CNS-0614957) and the Semiconductor Research Corporation (2005-HJ-1331).

Authors' address: Department of Computer Science and Engineering, University of California, Riverside, CA 92521; email: gstitt@cs.ucr.edu.

1. INTRODUCTION

Over the past two decades, numerous high-level synthesis techniques have been proposed as an alternative to the complex task of specifying register transfer-level (RTL) descriptions of hardware for RTL synthesis. In contrast to RTL synthesis, which automatically converts cycle-by-cycle hardware behavior descriptions to circuits, high-level synthesis converts algorithms to circuits. Due to the popularity of the C/C++ programming language, many C-based languages and high-level synthesis approaches [Böhm et al. 2002; Celoxica 2006; Fin et al. 2001; Fleury et al. 2001; Gajski et al. 1992; Gokhale and Stone 1998; Gupta and De Micheli 1992; Ku and De Micheli 1990; Najjar et al. 2003; Oxford 1997] have been proposed to partition descriptions among programmable processors (software) and custom processors (hardware) and to synthesize that hardware. Other approaches have proposed to synthesize and partition standard ANSI C [Athanas and Silverman 1993; Guo et al. 2004; Gupta et al. 2003; Mentor Graphics 2006; Tensilica 2006].

Despite the existence of many successful RTL synthesis tools [Synopsys 2006; Synplicity 2006], to our knowledge there are no widely successful high-level synthesis tools. One possible reason that high-level synthesis has not yet become highly successful is that these approaches typically impose restrictions on tool-flow that may be unattractive to many software developers. The requirement of using a nonstandard C language variation may be a significant deterrent for many software developers who are typically very reluctant to change languages. Other developers may instead prefer to use a language other than C, such as Java or Matlab.

Synthesis approaches that synthesize from standard C also impose restrictions by requiring a specific compiler or development environment. Software developers typically have preferred compilers and are likely to resist a change to well-established software development tools. One could argue that adding synthesis capabilities to popular compilers could solve this problem. However, there is little incentive for compiler developers to add synthesis capabilities because only a small percentage of the compiler users would be interested in synthesis. Therefore, a synthesis tool with a more transparent tool-flow integration is likely to be more widely accepted than traditional synthesis approaches.

We proposed binary synthesis in Stitt and Vahid [2002] to achieve a more transparent synthesis tool-flow by supporting all source languages and compilers of a particular microprocessor. Binary synthesis achieves this transparency by using the same tool-flow as software development for that microprocessor. A software developer first compiles an application into a software binary, and then uses an additional backend tool to partition and then synthesize the application. A drawback to such an approach is that the hardware synthesized from a software binary may be less efficient than hardware synthesized from high-level code. However, previous work has shown that, in many cases, binary synthesis achieves comparable hardware to high-level synthesis [Stitt et al. 2005a; Stitt and Vahid 2003, 2002; Stitt et al. 2005b].

In addition to achieving a more transparent tool-flow, binary synthesis has other advantages compared to traditional approaches. Binary synthesis can

generate hardware for legacy applications for which the source code is no longer available and for hand-optimized assembly code, which is common in many embedded systems and digital signal processing (DSP) applications. Binary synthesis can also generate hardware for library code which traditional approaches are unable to do because libraries are generally provided in object code format.

In this article, we discuss existing binary synthesis approaches and highlight the underlying techniques that make binary synthesis comparable to high-level synthesis. Section 2 describes the tool-flow of binary synthesis. Section 3 discusses related work. Section 4 introduces decompilation techniques used in binary synthesis approaches. Section 5 describes hardware/software partitioning techniques used in binary synthesis. Section 6 discusses synthesis techniques. Section 7 discusses techniques to modify a binary to support communication with synthesized hardware. Section 8 gives an overview of existing binary synthesis approaches.

2. BINARY SYNTHESIS

The tool-flow for binary synthesis is shown in Figure 1(a). The initial steps of binary synthesis are the same as any software development tool-flow. Initially, a software developer specifies an application in a high-level language and then compiles and links the high-level code into a software binary. Binary synthesis then creates hardware for critical regions of the binary and leaves regions not suitable for hardware in their existing software implementation. When targeting a field-programmable gate array (FPGA), binary synthesis outputs a bitfile that can be used to program a platform consisting of a microprocessor and FPGA. Binary synthesis can also target FPGA-only systems by either synthesizing hardware for the entire application or by synthesizing a soft-core microprocessor such as the Xilinx MicroBlaze [Xilinx 2006] or Altera NIOS [Altera 2006] to execute the software regions. Binary synthesis could conceptually also be used for ASIC (application-specific integrated circuit) flows, although ASIC designers are more likely to accept modified tool flows and are thus more amenable to traditional high-level synthesis methods.

Figure 1(b) illustrates the implementation of a typical binary synthesis approach. Decompilation initially recovers high-level information that is needed for synthesis and partitioning. Hardware/software partitioning then divides the decompiled representation into hardware and software regions. Next, synthesis converts the hardware regions into circuit netlists. Binary updating modifies the original software binary to use the synthesized hardware. The final step of binary synthesis combines the hardware netlists and modified binary into a bitfile that programs the targeted platform.

Note that although we call the approach binary synthesis, neither the synthesis nor partitioning is actually performed at the binary level. Because both partitioning and synthesis are performed on the decompiled representation, any existing high-level partitioning/synthesis technique may be used while still obtaining the more transparent tool-flow obtained by first compiling and then decompiling the original high-level code.

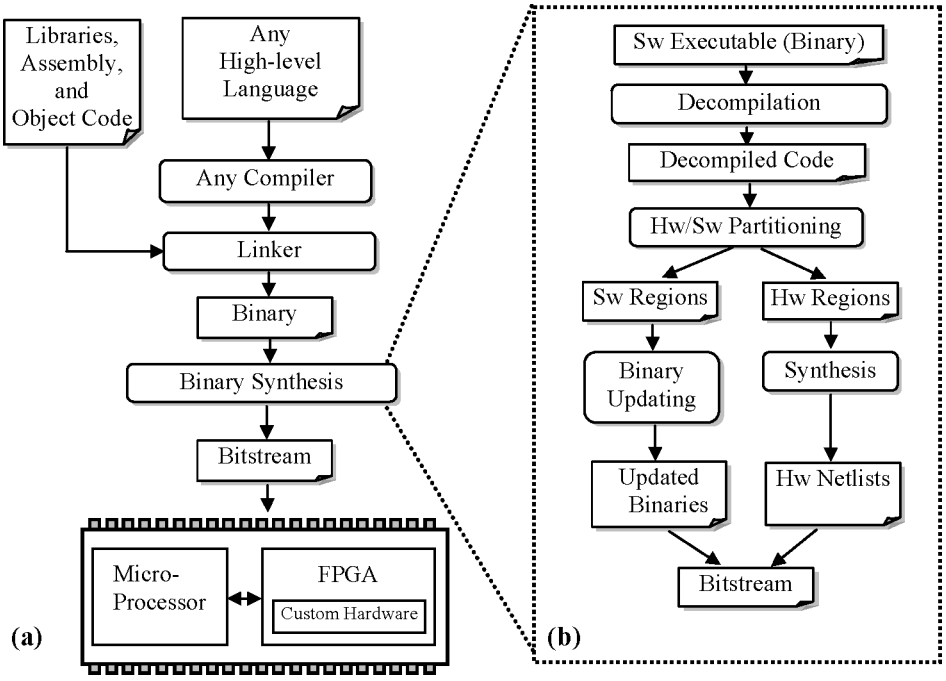

Fig. 1. Binary synthesis. (a) tool-flow. (b) implementation of a typical binary synthesis approach.

Although complete synthesis from a software binary is possible if the entire application is appropriate for hardware, most applications typically benefit from leaving certain regions in software. For many regions, there is either no performance benefit from a hardware implementation or the area requirements of a hardware implementation are too large. A common example is the initialization of a global array of constants. In software, this code would consist of a series of instructions that store the constants to the appropriate memory location. In hardware, storing these constants in FPGA lookup tables would require a large overhead. Also, additional logic may be needed to transfer these constants into on-chip block RAMs or off-chip memory. Clearly in this situation, a software implementation is more efficient. Depending on the constructs used in the original high-level code, the entire application may be synthesized to hardware or, in situations where the code uses non-hardware-suitable constructs, the hardware regions may be limited to several computational kernels.

3. RELATED WORK

There are a number of fields of research that are related to binary synthesis. In this section, we describe approaches orthogonal to binary synthesis: C-based languages and synthesis, ASIPs, and dynamic software optimization and binary translation. These techniques also address the problem of improving software performance with minimal changes to tool-flow. In later sections, we discuss previous work utilized by binary synthesis: decompilation techniques in

Section 4.1, hardware/software partitioning in Section 5.1, synthesis in Section 6.1, and binary updating techniques in Section 7.1.

3.1 C-Based Languages and Synthesis Approaches

Previous efforts have attempted to simplify the specification of RTL hardware by creating C-based hardware description languages with high-level constructs. Other works have introduced synthesis approaches from C-based languages. The motivation behind these approaches is that C is a popular high-level language and therefore new hardware languages or synthesis approaches based on C are more likely to be accepted.

One of the early attempts at C-based languages for synthesis was the language NapaC [Gokhale and Stone 1998], which used standard C code with pragma directives to specify the data and computation to be implemented in hardware. HardwareC [Ku and De Micheli 1990] is a C-based hardware description language that uses process constructs much like VHDL, but uses C syntax to describe each process. Streams-C [Frigo et al. 2001] is a hardware description language that consists of a subset of standard C, a set of libraries and intrinsic functions, and a communicating process model. Handel-C [Celoxica 2006; Fleury et al. 2001; Oxford 1997] is a C-like language that is strongly typed and contains constructs for explicitly specifying parallel constructs similar to the occam language, while excluding C constructs that are not always appropriate for hardware such as floating-point operations, integer divide, etc. SystemC [Fin et al. 2001] is a language that uses pure C++ syntax but contains constructs that are more appropriate for hardware specification. By using C++ syntax, SystemC can be simulated in software by compiling it with any C++ compiler and then linking in the SystemC library. In addition, SystemC allows standard C constructs, enabling easier hardware/software cosimulation. SA-C [Böhm et al. 2002; Najjar et al. 2003] is a single assignment C-based language that contains constructs for specifying window operations that are common in DSP and image processing applications.

In addition to the introduction of C-based languages, there have been numerous approaches that synthesize from standard C. PRISM [Athanas and Silverman 1993] was an early C-based synthesis approach that synthesized standard C code onto multiple FPGAs. The Nimble compiler [Li et al. 2000] was a commercial partitioner from Synopsys that utilized dynamic reconfiguration to swap in hardware regions synthesized from C. Celoxica's DK design suite [Celoxica 2006] allows applications to be specified in both C and Handel-C for regions to be synthesized, and then performs partitioning and verification. DEFACTO [Diniz et al. 2005], ROCCC [Guo et al. 2004], and SPARK [Gupta et al. 2003] synthesize standard C code while applying parallelizing compiler transformations.

Previous C-based languages and synthesis approaches represent good technical solutions for synthesis from high-level code. The main goal of these approaches is to maximize performance with a lessened impact on tool-flow compared to traditional hardware description languages such as VHDL or Verilog. Binary synthesis is not meant to be a replacement for these approaches but is

instead intended to be a complementary approach that greatly reduces tool-flow integration at the potential cost of slightly degraded hardware performance. Binary synthesis mainly targets software developers who are not willing to use a nonstandard compiler and a nonstandard language.

3.2 Application-Specific Instruction Set Processors (ASIPS)

ASIPs (Application-Specific Instruction Set Processors) improve the performance/power of an application compared to a standard microprocessor by adding custom instructions that are tuned to a specific application. Kucukcakar [1999] introduced a methodology for developing new instructions which involved profiling to determine frequent instruction sequences and then replacing these sequences with custom instructions implemented in configurable logic. Fisher [1999] discussed customizing VLIW instructions sets to a particular domain of applications as opposed to a single application.

These previous ASIP approaches achieve good results but have problems with tool-flow because of the requirement of developing a new Compiler, simulator, profiler, etc. Tensilica [Gonzalez 2000] reduces some of the tool-flow impact by automatically generating an entire tool-flow based on custom instructions defined in a specialized language called TIE (Tensilica Instruction Extension). The XPRES compiler [Tensilica 2006] simplifies the creation of ASIPs by analyzing C code and determining custom instructions, eliminating the need to manually define TIE instructions. Stretch [2006] is a similar approach that creates custom instructions in an FPGA instead of ASIC, allowing for the instructions to be reconfigured.

One limitation of ASIP approaches is that although they reduce tool-flow integration problems by automatically generating the tool chain, the software designer is still limited to using those generated tools. Many software developers may prefer to use a different compiler or even a different language. Recently, transparent ASIP approaches have been proposed that create custom instructions at runtime by reconfiguring a matrix of functional units to create a custom hardware circuit for instructions within the fill unit of a trace cache [Clark et al. 2005, 2004].

Binary synthesis has similar transparency advantages as runtime ASIP approaches, theoretically supporting all compilers and languages. In addition, binary synthesis likely achieves better performances than runtime ASIP approaches by not being limited to creating hardware for instructions in the fill unit of the trace cache. Binary synthesis can achieve similar or greater performance improvements compared to static ASIP approaches with the tool-flow advantages of runtime ASIP approaches.

3.3 Dynamic Software Optimization and Binary Translation

Approaches for dynamic software optimization and binary translation have been proposed to maintain binary compatibility and to reduce compilation time. Dynamo [Bala et al. 2000] is a dynamic optimization approach that profiles an application during execution to determine frequent paths, optimizes the code for those paths, and stores the optimized code in a special fragment

cache. When software execution reaches a frequent path, the microprocessor fetches instructions from the fragment cache to execute the optimized code. FX!32 [Chernoff et al. 1998] dynamically translates x86 binaries into Alpha binaries by first emulating the application and profiling to determine frequent regions that should be translated to native Alpha instructions. Daisy [Ebcioglu et al. 2001] is a similar approach that first profiles using a series of 8k 8-way cached counters and then dynamically converts PowerPC binaries into VLIW code to maintain compatibility and to exploit parallelism in the code. BOA [Gschwind et al. 2000] dynamically translates PowerPC instructions into smaller microinstructions that can be more easily pipelined and scheduled in parallel. BOA also detects frequent paths, performs path-specific optimizations and translates paths from PowerPC code into native VLIW code. The IA-32 execution layer for the Itanium microprocessor uses software to convert IA-32 instructions into native Itanium instructions [Baraz et al. 2003]. Instruction path coprocessors [Chou and Shen 2000] dynamically optimize regions of code that are read into the fill unit of a trace cache. A similar approach [Friendly et al. 1998] performs path-specific optimizations on regions in the fill unit of a trace cache and translates the optimized paths to a different underlying architecture.

Binary translation is also becoming a more common technique used in computer architecture to support an underlying architecture that may be different than the instruction set architecture. Pentium architectures efficiently execute x86 binaries by first translating instructions to microinstructions that are more easily pipelined and parallelized. The Transmeta Crusoe [Dehnert et al. 2003] and Efficeon [Transmeta 2006] perform code morphing to dynamically convert x86 binaries into native VLIW instructions that save power compared to Pentium processors.

Many of these dynamic software optimization approaches have the same goal as binary synthesis which is to achieve a more transparent optimization of an application. Binary synthesis differs mainly by synthesizing hardware, achieving much greater performance improvements compared to performing software optimizations. One potential advantage of dynamic software optimization is the ability to use runtime information, such as phases or frequent values, to further optimize the code. Although most binary synthesis techniques are static, warp processing [Lysecky et al. 2006; Lysecky and Vahid 2003; Stitt et al. 2003] has demonstrated the feasibility of performing binary synthesis at runtime, allowing binary synthesis to also take advantage of runtime information to optimize hardware.

4. DECOMPILATION

Decompilation is perhaps the most important task in binary synthesis. We define *decompilation* as the recovery of high-level information from a low-level software representation such as from a binary or assembly code. Other definitions exist, such as the conversion of assembly code into a specific high-level language. Decompilation is also commonly referred to as reverse compilation or reverse engineering. We use a more general definition to include the many

low-level and high-level representations used by different binary synthesis approaches. The low-level representation may be a software binary, assembly code, or object code. High-level representations may include a specific language such as C, an abstract syntax tree, a control/data flow graph (CDFG) annotated with high-level constructs, or a machine syntax tree.

Early efforts in binary synthesis performed very limited decompilation, generally only recovering a control/data flow graph representation of the binary. Synthesis from this partially decompiled representation typically resulted in speedups from 1.2x to 2x compared to software execution on a 100 MHz MIPS. The reason for the small performance improvement was that the parallelism of the synthesized hardware was limited to the instruction-level parallelism of the assembly code. Therefore, binary synthesis without decompilation rarely achieves better performance than a VLIW or multi-issue microprocessor.

Decompilation is therefore a necessary step for effective binary synthesis. As an example, consider that for many applications, synthesis exposes parallelism by unrolling loops. Without knowledge of the loop structures and bounds, the parallelism visible to a binary synthesis approach is very limited.

4.1 History of Decompilation

In this section, we briefly summarize previous decompilation approaches. For a more complete summary, we refer the reader to Cifuentes [1994] and Decompilation [2006].

Initial decompilation approaches appeared between the mid-1960s and the 1980s. The motivation behind the majority of early decompilation approaches was to port legacy binaries to newer machines by recovering the high-level source and then recompiling it, [Halstead 1967; Sassaman 1966]. In some situations, decompilation was also used for converting between languages by first compiling from a source language and then decompiling into the target language [Barbe 1974; Housel and Halstead 1974]. Decompilation also proved useful for documenting and maintaining applications written in assembly by decompiling, making modifications, and then recompiling [Hopwood 1978]. Software developers have also used decompilation as a debugging tool for assembly code by recovering a high-level representation that is easier to check for errors [Barbe 1974].

Early decompilation approaches typically had several limitations. Some approaches could not actually decompile a binary and had to decompile object or assembly code—a far simpler task that didn't require separating code and data and that also made control structures easier to recover [Hollander 1973; Sassaman 1966; Workman 1978]. Many approaches performed limited control/data flow analysis, resulting in an inefficient decompiled representation [Friedman and Schneider 1973; Hopwood 1978; Housel and Halstead 1974]. Some decompilation approaches were limited to working on binaries compiled from a specific compiler [Hollander 1973; Schneider and Winiger 1974]. Other approaches were unable to recover high-level code [Hopwood 1978] or could only recover code in a specific language [Halstead 1970; Housel and

Halstead 1974]. In addition, most initial approaches could not recover useful data types.

Later decompilation approaches have utilized decompilation to port applications to a different architecture or to optimize an application for a newer version of a similar architecture [Brinkley 1981; Sites et al. 1993]. More recent approaches still use decompilation for understanding/maintaining legacy code, debugging, and verifying the correctness of a compiled application [Hood 1991]. In addition, decompilation has recently been used to verify security of an application and for the detection of viruses in a binary [Cifuentes et al. 2001; Hood 1991].

More recent approaches from the 1990s have removed many of the limitations from earlier approaches. Decompiler compilers [Breuer and Bown 1994] attempted to automate the generation of a decompiler for any compiler but were limited to compilers for which detailed specifications were given. The 8086 decompiler [Fuan and Zongtian 1991] was one of the first attempts to recover arrays, pointers, and data types. In addition, this decompiler recognized library functions, resulting in more readable C code. Cifuentes [Cifuentes 1994; Cifuentes and Van Emmerik 2000; Cifuentes et al. 1999] described formal procedure and control structure recovery techniques in addition to a formal method for specifying machine semantics, allowing for easier porting of decompilers to different instruction sets. In Mycroft [2001, 1999], the author presented a method for the recovery of data types.

In the 1990s, decompilation techniques were also used for performing binary translation. UQBT (University of Queensland Binary Translation) [Cifuentes and Van Emmerik 2000] is a retargetable binary translater that uses decompilation techniques to convert a binary into a high-level intermediate representation that can then be optimized and translated into a binary of a different instruction set.

Over the past decade, several research-based decompilers have appeared. The DisC decompiler [DisC 2006] decompiles to C code but only works on binaries generated from the TurboC compiler. DCC [Cifuentes 1994] decompiles x86 binaries to C code but is limited to binaries under 30 kilobytes. To our knowledge, one of the most complete decompilers is Boomerang [2006], which is an open-source retargetable decompiler that currently supports x86, SPARC, and PowerPC binaries. REC (Reverse engineering compiler) [REC 2005] is a similar approach that supports x86, PowerPC, and MIPS R3000 binaries, in addition to supporting a variety of operating systems.

Recently, Java decompilation has received attention because of the usefulness of decompilation for understanding/debugging third-party class libraries. Also, Java byte code contains class information, thread information, and type information that is not available in other binaries, making Java byte code much easier to decompile. Recent approaches such as Mocha [1996], JReverse Pro [2006], and SourceAgain [Ahpah 2004] can successfully decompile Java byte code, sometimes recovering a high-level representation that is identical to the original code. Java decompilation has proven to be so successful that obfuscators have begun to appear to help protect the security of Java applications by increasing the difficulty of decompilation [Chan and Yang 2004].

4.2 Summary of Conventional Decompilation Techniques

In this section, we describe conventional decompilation techniques and the structure of a decompiler, which is similar to the structure of a compiler. The process of decompilation can be divided into three phases. In the first phase, which we refer to as intermediate code creation, the front end of the decompiler parses the binary to create an intermediate representation, which is analogous to a compiler parsing the high-level source code to create an abstract syntax tree. The next phase is high-level construct recovery, during which the decompiler analyzes the intermediate code to determine control structures such as loops and if statements, and data structures such as arrays. This phase is analogous to compiler optimizations applied on intermediate code. The final phase, or the back end, generates a high-level representation by mapping the intermediate code and high-level constructs into statements and expressions from the targeted high-level language. This phase is similar to the back-end of a compiler, where the intermediate code is mapped onto assembly instructions.

In this paper, we discuss the first two phases of decompilation. We omit details on the decompiler back-end because generating high-level source code is unnecessary for most binary synthesis approaches, which typically synthesize at the level of a control/data flow graph or syntax tree.

4.2.1 *Intermediate Code Creation.* Figure 2 illustrates the intermediate code creation phase of decompilation, showing at each step how the decompiled representation changes. The C code and corresponding assembly to be decompiled are shown in Figure 2(a). This code implements a simple function that accumulates all values in an array and then returns the accumulated value.

Binary parsing is the first step of intermediate code generation. During binary parsing, the decompiler analyzes the software binary, determines the instructions and data in the binary, and then creates a parse tree for each instruction. The main difficulty in this step is the separation of code and data, which is equivalent to the halting problem. However, previous efforts have introduced heuristics that can separate code and data with a high success rate [Cifuentes and Van Emmerik 2000]. These heuristics start at the first instruction, which is assumed to be at the beginning of the binary, and then progressively determine all other instructions by following the control flow of each subsequent instruction. Alternatively, if symbol information is included in the binary, then separating code and data is trivial.

After separating code and data, the decompiler converts each instruction into an instruction-set independent representation. A commonly used intermediate representation in decompilers is register-transfer lists [Cifuentes 1994]. A register transfer is a statement that defines a particular register. During this process, the decompiler makes all instruction side effects explicit using register transfers. For example, a pop instruction would use two register transfers: one to define the value being read from the stack and another to define the stack pointer. Register transfers are also used to define condition flags that may be set by an instruction. Figure 2(b) shows the register transfers created during binary parsing for the given example.

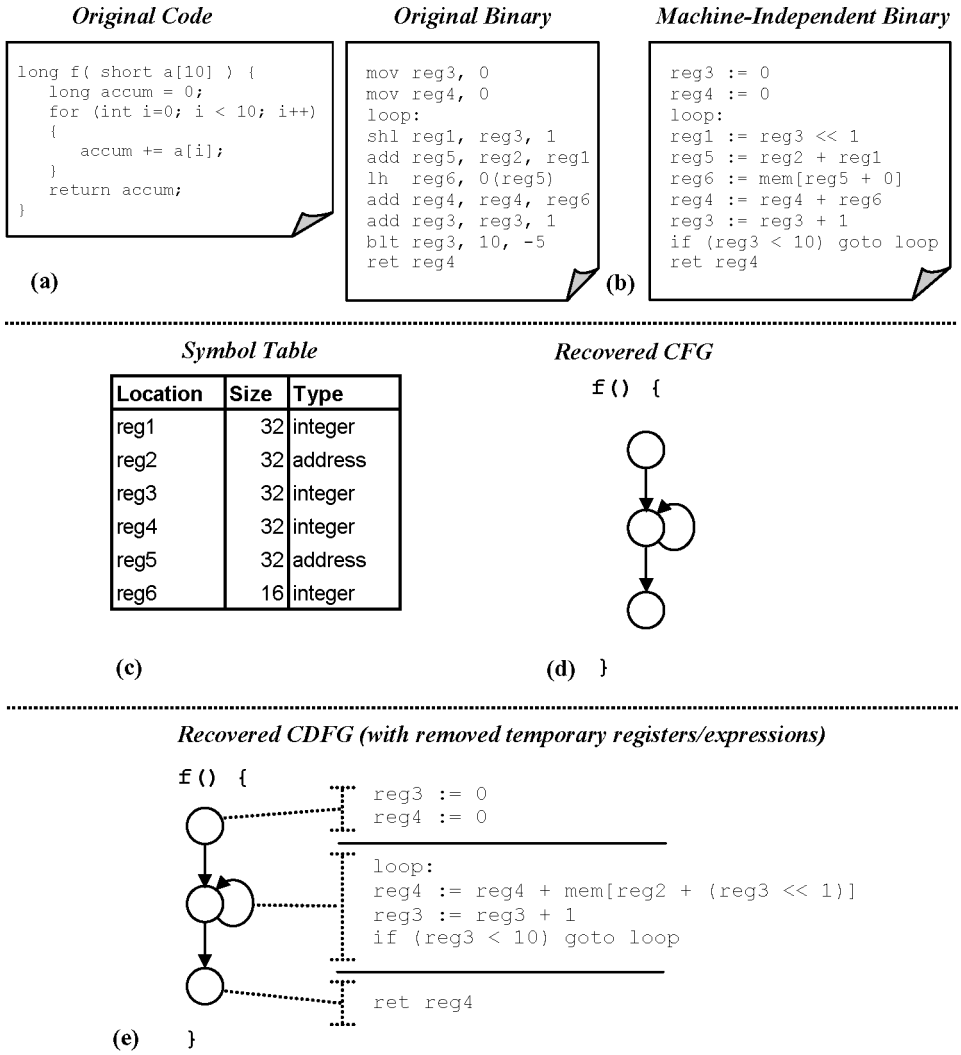

Fig. 2. Decompilation of a simple function into a CDFG. (a) The original C code and corresponding assembly. (b) The machine-independent representation (register transfers) created by binary parsing. (c) The symbol table created during type recovery. (d) The recovered CFG. (e) The recovered CDFG after removing temporary registers and expressions.

Type recovery analyzes the uses of data in the register transfer lists to determine low-level types. All data at the binary level is either an integer or a float which is specified by the semantics of each instruction. Integer types can be further classified into addresses of instructions and addresses of data. Type recovery determines addresses to data by analyzing registers that are used in load or store instructions. Type recovery determines addresses to instructions by analyzing indirect jump instructions. Type recovery also determines the size of data from instructions that assign an immediate to a register or instructions that load a specific size (load word, load byte, etc.). Logic optimization also

determines data size in the situation when certain bits may be ended with zeros. After determining the type and size of data, type recovery uses definition-use analysis to propagate this information to all other instructions, further refining the types.

Figure 2(c) illustrates type recovery for the given example. Type recovery determined that registers 1, 3, 4, and 6 were integer types and registers 2 and 5 were addresses. All values in the registers were determined during type recovery to be 32-bits wide except for register 6 which was 16-bits wide.

The decompiler next performs control flow graph (CFG) recovery by performing basic block analysis to determine the nodes of the CFG. These nodes may consist of 1-way nodes (jumps), 2-way nodes (conditional branches), n-way nodes (indirect jumps), call nodes (function calls), return nodes, and fall-through nodes. Next, the decompiler connects these nodes based on the targets of the control instructions to form a complete control flow graph. Figure 2(d) illustrates control flow graph recovery for the given example. The control flow graph consists of three basic blocks: one block to initialize the induction variable and accumulated value, one block that represents the body of the loop, and one block that returns from the function.

After recovering the control flow graph, the decompiler analyzes the register transfer lists and recovers a data flow graph for each function by performing intraprocedural definition-use analysis on the register transfer lists. After creating the data flow graph, data flow analysis optimizes the data flow graph to eliminate inefficiencies introduced by the instruction set and assembly code such as temporary registers and temporary expressions. In addition, data flow analysis eliminates many register transfers that correspond to condition flags that are guaranteed to never be set. Figure 2(e) shows the decompiled control/data flow graph after data flow recovery and analysis. In this example, the decompiler has optimized away all registers except register 2, 3, and 4.

4.2.2 *High-Level Construct Recovery.* After generating the intermediate control/data flow graph representation of the binary, decompilation performs high-level construct recovery as shown in Figure 3. For simplicity, the figure shows the decompiled code as high-level C code. The actual representation is typically an annotated CDFG or abstract syntax tree.

The first step of high-level construct recovery is function recovery, which analyzes the targets of call instructions or jump-and-link instructions to determine function calls. Next, function recovery analyzes inputs to the data flow graph for each function to determine function parameters that are passed in registers. Some decompilation techniques also check common parameter-passing conventions to determine parameters passed on the stack [Cifuentes 1994]. Figure 3(a) shows the decompiled function after function recovery.

Control structure recovery [Cifuentes 1994; Cifuentes and Van Emmerik 2000; Cifuentes et al. 1999] analyzes the control flow graph to determine high-level control structures such as loops and if statements. Control structure recovery determines loop structures using interval analysis. An interval contains a maximum of one loop which must start at the head of the interval. After checking all the intervals of a control flow graph for loops, each interval is collapsed

Function Recovery

```
long f( long reg2 ) {
   long reg3 = 0;
   long reg4 = 0;
   loop:
   reg4 = reg4 + mem[reg2 + (reg3 << 1)];
   reg3 = reg3 + 1;
   if (reg3 < 10) goto loop;
   return reg4;
}
```

(a)

Control Structure Recovery

```
long f( long reg2 ) {
   long reg4 = 0;
   for (long reg3 = 0; reg3 < 10; reg3++) {
      reg4 += mem[reg2 + (reg3 << 1)];
   }
   return reg4;
}
```

(b)

Array Recovery

```
long f( short array[10] ) {
   long reg4 = 0;
   for (long reg3 = 0; reg3 < 10; reg3++) {
      reg4 += array[i];
   }
   return reg4;
}
```

(c)

Fig. 3. High-level construct recovery from a CDFG (Figure 2(e)). The decompiled function is shown after (a) function recovery, (b) control structure recovery, and (c) array recovery, which is almost identical to the original code from Figure 2(a).

into a single node, forming a new control flow graph which is then checked for additional loops. This process is repeated until the control flow graph can no longer be reduced. By processing loops in this order, control structure recovery can determine the proper nesting order of loops. After finding a loop, control structure recovery determines the type of the loop as pretested, posttested, endless, or multi-exit, based on the location of the exit from the loop. Control flow and data flow analysis determines loop induction variables that the decompiler can then analyze to determine loop bounds.

For brevity, we omit a description of determining if statements. Cifuentes gives a complete description of the determination of if statements in Cifuentes et al. [1999]. Figure 3(b) shows the decompiled function after recovering control structures.

Array recovery [Cifuentes 1994; Stitt et al. 2005a] analyzes memory accesses of loops to determine linear patterns which correspond to array accesses. Nonlinear patterns can also be analyzed but are generally omitted because linear patterns are more common. After identifying an array with linear memory access patterns, array recovery determines the size of the array from the bounds of the loop that accesses the array. If a loop is unbounded or the bounds cannot be determined, then array recovery is unable to determine the size of the

array. The size determined from the loop bounds may only represent a subset of the array in the situation that the loop does not access the entire array. Therefore, after determining arrays from loops, array recovery uses heuristics to map memory accesses outside of the loop onto the recovered arrays. If array recovery is unable to determine all elements of the array, the missing elements are treated as separate variables.

To recover multidimensional arrays, array recovery analyzes nested loops to detect row-major ordering or column-major ordering calculations. The size of each dimension corresponds to the bounds of each nested loop. Multidimensional array recovery is generally less successful than single-dimension array recovery because of the multiple ways that a compiler may implement row-major and column-major ordering calculations.

Figure 3(c) shows the final decompiled function after recovering arrays. Note the similarity between the decompiled code and the original C code in Figure 2(a). The only difference is that the decompiler replaced variable names with the corresponding registers to which the compiler mapped the variables.

4.3 Limitations

Decompilation is not always as successful as was illustrated in Figures 2 and 3. Indirect jumps may result in failure to decompile a specific region, or even the entire application if the decompiler cannot separate code and data. Nonlinear memory access may result in the decompiler failing to recover arrays. However, despite these limitations, previous work has shown that decompilation performed during binary synthesis can almost always recover enough high-level information to synthesize efficient hardware [Stitt et al. 2005a; Stitt and Vahid 2003; Stitt et al. 2005b] for embedded system applications and DSP applications. One reason for this high success rate is that applications in these domains tend to be written using constructs that are ideal for decompilation.

4.4 New Decompilation Techniques for Binary Synthesis

Most existing decompilation techniques were developed to recover a high-level representation of an application for purposes of maintenance, debugging, and translation. Although decompilation commonly recovers high-level code similar to the original code, software compiler optimizations applied to the original code may transform the binary representation, resulting in a different decompiled representation. Although the recovered code is different than the original code, this representation is generally suitable for the traditional uses of decompilation.

When performing decompilation for the purposes of binary synthesis, the recovered representation must be appropriate for generating fast hardware. Optimizations applied to the code by the software compiler may result in a representation that is unsuitable for hardware. Software compiler optimizations tend to force the synthesis tool into a particular implementation, which often does not improve the hardware, especially considering that the compiler optimizations target a microprocessor architecture with completely different resources than would be used in custom hardware. Therefore, binary synthesis

approaches require new additional synthesis-specific decompilation techniques to recover the original unoptimized representation. Although unoptimizing a program may seem counterintuitive, the idea is to allow the synthesis tool to make all optimization decisions. Also, removing optimizations should not result in decreased performance because the synthesis tool can always perform the optimizations again.

Loop unrolling is an example of a software compiler optimization that can be problematic for binary synthesis. Unrolled loops may result in less detailed profiling information. Many critical region detection techniques profile backwards-branch instructions to determine loop execution times. If a compiler completely unrolls a loop, then these backwards-branch instructions will not be present the binary. Also, loop unrolling can remove or obscure memory access patterns that are important for array recovery and synthesis of advanced memory structures such as smart buffers [Guo et al. 2004]. Loop unrolling may also greatly increase the size of the control/data flow graph, resulting in significant increases in synthesis execution times due to superlinear complexity synthesis heuristics, which may make a binary synthesis approach less practical. Also, even if a synthesis tool unrolled a loop to expose parallelism, the amount of unrolling applied by the software compiler is unlikely to match the amount of unrolling applied by the synthesis tool.

To deal with the problems of loop unrolling, a new decompilation technique, *loop rerolling*, was introduced in Stitt and Vahid [2005b]. Loop rerolling first detects instances of loop unrolling in the software binary by using efficient pattern detection and definition-use analysis, and then rerolls the unrolled iterations of the loop into a single iteration that more closely represents the loop in the original code.

Strength reduction is another software compiler optimization that is potentially problematic for binary synthesis. Typically, a compiler applies strength reduction to convert an expensive multiplication into a series of shift-and- add operations. This optimization is more effective when one of the inputs of the multiplication is a power of 2 or close to a power of 2. However, in several situations, strength reduction may result in less efficient hardware. Converting a multiplication into a series of shifts and adds may exhaust all adder resources even if multipliers are available. In this situation, strength reduction forces the binary synthesis tool into a particular implementation using adders, whereas using a multiplier might have produced better hardware. Also, some microprocessors, such as a MicroBlaze, may not have a multiplier, requiring the software compiler to convert all multiplications into shifts and adds. Clearly, in this situation, binary synthesis would generate more efficient hardware if the compiler had not performed any strength reduction.

A new decompilation technique, strength promotion, was introduced in Stitt and Vahid [2005b] to deal with the problems of strength reduction in binary synthesis. Strength promotion detects instances of strength reduction within the binary and then promotes the reduced operations into a form likely to match the original form. Strength promotion does not hurt hardware performance because a synthesis tool can always perform the strength reduction again after determining the available resources. Although promoting and then reducing

seems redundant, promoting the operations prevents the synthesis tool from being forced into a less-efficient adder implementation when multipliers are available.

Other synthesis-specific decompilation techniques consist of applying existing optimizations to decompiled code. One such optimization eliminates inefficiency caused by compare-with-zero instructions. A compare-with-zero instruction is a control instruction that results in a change to the program counter, based on the comparison of a specific register with the constant value zero. To implement comparisons with nonzero values, an additional instruction performs an arithmetic operation, typically subtraction, of the values compared. If not removed, these arithmetic instructions may result in many unnecessary subtractors in the synthesized hardware. Decompilation in binary synthesis performs optimizations that replace subtractions followed by compare-with-zero instructions with a simple comparison of two values.

Future dynamic binary synthesis approaches may require additional specialized decompilation techniques that trade off high-level information for reduced decompilation time. Existing dynamic approaches utilize standard decompilation techniques because the time required for placement and routing dominates the execution time of binary synthesis.

5. HARDWARE/SOFTWARE PARTITIONING

Although binary synthesis is capable of converting a software binary completely to hardware, a hardware-only implementation may not be efficient. For some regions of an application, hardware may yield little or no performance benefit compared with software. Implementing those regions in hardware may yield larger than necessary hardware; software on a microprocessor may be a more size-efficient implementation. Thus, modern synthesis solutions, whether high-level or binary might use hardware only for the regions that yield large speedups when in hardware and use software for the remaining regions. Hardware/software partitioning identifies performance-critical regions of an application, and then decides which of these regions should be implemented in hardware [Chiodo et al. 1994; Gajski et al. 1994; Wolf 1994].

5.1 Summary of Hardware/Software Partitioning Techniques

Hardware/software partitioning approaches can be classified into two application categories, sequential and parallel. Partitioning of sequential applications deals with moving critical loops, blocks, and functions into hardware. Partitioning of parallel applications deals with mapping tasks or processes onto multiple microprocessors or hardware coprocessors.

Hardware/software partitioning consists of five subproblems: choosing a region granularity, evaluating partitions using implementation and estimation techniques, consideration of different implementations of each region, consideration of different implementation models, and exploration of the partition solution space which is guided by profiling.

Hardware/software partitioning initially selects a region granularity to consider for hardware implementation. Selecting region granularity involves

analysis of the trade-offs between fine-grained and coarse-grained granularities. Fine-grained granularities, such as blocks or statements, provide flexibility for selecting code for hardware but result in a large number of partitions that may require long exploration times. Coarse-grained granularities, such as loops and functions, reduce the number of possible partitions but may contain code that is not appropriate for hardware. Recently, approaches have also considered a granularity of paths, which potentially combine the reduced partitions of loops and functions with the flexibility of more fine-grained approaches. Path granularity has the disadvantage of a runtime overhead to check if the appropriate path is taken. Parallel partitioning approaches also consider the more coarse-grained granularities of tasks or processes which may be decomposed into more fine-grained granularities.

To determine a solution, hardware/software-partitioning approaches perform partition evaluation by evaluating design metrics such as hardware and software execution time, communication time, hardware area, power, energy, etc. Parallel approaches also consider the effects of scheduling multiple tasks onto shared processors. Evaluation approaches may determine actual values of design metrics by synthesizing or compiling regions, or may trade off accuracy to reduce execution time by using estimation techniques [Enzler et al. 2000; Kannan et al. 2002; Kulkarni et al. 2002; Shayee et al. 2003; Xu and Kurdahi 1996].

Another issue related to hardware/software partitioning is the consideration of multiple implementations of each region. For example, a region consisting of an accumulate loop could be implemented using a single adder or could alternatively be unrolled and implemented as a large adder tree. Other possibilities include different amounts of unrolling, pipelining, etc. Partitioning approaches must consider the trade-off of evaluating multiple implementations and keeping the total number of possible partitions at a reasonable size.

In some cases, hardware/software partitioning must choose between different implementations models for hardware regions. One issue is deciding between a mutually exclusive or parallel execution model for microprocessors and coprocessors. Hardware/software partitioning also considers different communication models, such as a shared register file [Jones et al. 2005], shared memory, shared cache, and memory-mapped registers [Stitt and Vahid 2005a], which represent different trade-offs of latency and bandwidth.

Finally, hardware/software partitioning performs exploration of the partition solution space to minimize an objective function or to meet constraints. Partition exploration methods were introduced in the early 1990s by the digital-hardware design automation community [Ernst and Henkel 1992; Gupta and De Micheli 1992; Vahid and Gajski 1997]. These early techniques were based on digital circuit partitioners that had also come from that community and had matured by that time. As opposed to partitioning basic circuit components, such as transistors, early hardware/software partitioning methods started with the basic statements or operations of an application and then applied powerful circuit partitioning algorithms to examine thousands of possible partitions of these statements/operations in search of the best partition. Many of the subsequent exploration methods have also been based on circuit partitioning, utilizing

sophisticated algorithms like min-cut [Vahid 1997], dynamic programming [Ernst and Henkel 1992], simulated annealing [Eles et al. 1997], genetic evolution [Srinivasan et al. 1998], and tabu-search [Eles et al. 1997].

Partition exploration approaches are guided by profiling techniques, which either instrument the code with profiling instructions [Duesterwald and Bala 2000] or simulate the application while monitoring special instructions that correspond to high-level constructs. These profiling techniques all determine performance-critical regions after the application has executed. Alternatively, some approaches detect performance-critical regions during execution by using architectural support for monitoring instructions. The M*CORE processor [Scott et al. 1999] profiles backwards-branch instructions to determine loops that should be placed in a special loop cache. Gordon-Ross introduced a more flexible nonintrusive hardware-based approach for detecting frequent loops [Gordon-Ross and Vahid 2003]. Ammons [Ammons et al. 1997] proposed an approach for using hardware counters found in architectures such as the Pentium 4 to perform profiling. Vaswani [Vaswani et al. 2005] introduced a nonintrusive hardware profiler for detecting hot paths.

5.2 Partitioning Techniques for Binary Synthesis

By decompiling to recover high-level constructs, binary synthesis might apply existing high-level hardware/software partitioning techniques for choosing a granularity, evaluating partitions, considering multiple region implementations, considering implementation models, exploring the partition solution space, and profiling. However, some specialized techniques have been introduced.

For static binary synthesis, certain granularities may not be considered if decompilation is unable to recover the corresponding high-level constructs. For example, if decompilation does not recover loops, then critical region detection can only consider functions and blocks. For profiling approaches by instrumentation of code, static binary synthesis must first decompile the binary to recover high-level constructs, insert code to profile those constructs, and then regenerate a binary using the binary update techniques discussed in Section 7.

Dynamic binary synthesis typically uses more specialized partitioning techniques due to the need to reduce partitioning times and memory usage to enable on-chip execution. Dynamic approaches generally consider coarse-grained granularities, such as loops and functions, to reduce the number of possible partitions. Similarly, dynamic approaches limit the amount of implementations considered for each region and the number of implementation models. For partition evaluation, dynamic binary synthesis utilizes fast estimation techniques, trading off accuracy to reduce the execution time required by actual implementations. Dynamic approaches also use specialized partition exploration algorithms that have much shorter execution times compared to traditional exploration techniques. Warp processors [Lysecky et al. 2006] perform a simplified partition exploration based on a greedy 0-1 knapsack heuristic. An on-chip profiler initially provides the exploration heuristic with a list of critical loops, sorted in order of percentage of execution time. The exploration heuristic then selects

loops for hardware implementation in sorted order until area is exhausted. The complexity of this heuristic is O(n), where n is the number of critical loops. Because the number of critical loops for most applications tends to be small, this heuristic requires exploration times several orders of magnitude shorter than traditional approaches and, in many cases, can achieve competitive solutions. For profiling, dynamic approaches may be more likely to rely on architectural support due to the requirement of determining critical regions at runtime. Profiling by instrumentation of code can be used by dynamic approaches but adds complexity due to introducing extra instructions, which may also reduce the performance of the application. Profiling using simulation is clearly not possible for dynamic binary synthesis. Determining critical regions after an application executes is possible but only for systems that repeatedly execute the same application.

6. SYNTHESIS

After hardware/software partitioning selects regions for hardware implementation, synthesis converts those regions into custom hardware circuits. Synthesis techniques can be divided into high-level techniques [De Micheli 1994; Gajski et al. 1992; Walker and Camposano 1991] that a custom controller and datapath for an algorithm, and low-level techniques that optimize the logic within the circuit in addition to optimize the layout and connection of components to maximize clock frequency [Betz and Rose 1997; Betz et al. 1999; Brayton et al. 1984].

6.1 Summary of Synthesis Techniques

High-level synthesis techniques initially perform lexical analysis and parsing of high-level code to convert the code into an intermediate representation such as an abstract syntax tree or control/data flow graph. Synthesis then performs standard compiler optimizations such as tree-height reduction, constant propagation, dead code elimination, etc.

After optimizing the intermediate code, scheduling determines the starting time of each operation in the code. Scheduling algorithms may be resource-constrained in which case the scheduler tries to minimize cycle latency without violating resource constraints such as a specific number of adders, multipliers, etc. Scheduling algorithms may also be performance-constrained in which case the scheduler attempts to achieve the specified performance while using minimal hardware resources. Other scheduling techniques examine Pareto-optimal solutions, trading off performance for reduced area, etc.

During scheduling, high-level synthesis performs resource allocation to determine the datapath resources in the synthesized hardware. Resource allocation typically allocates resources based on the maximum number of operations required in any cycle.

After allocating resources, high-level synthesis performs binding, which assigns operations from the code onto hardware resources, sometimes considering the sharing of resources among multiple operations. One goal of binding is typically to reduce steering logic such as muxes, buses, and registers.

Control synthesis creates a controller that configures the synthesized datapath to execute the appropriate scheduled operations each cycle.

Following high-level synthesis, low-level synthesis performs combinational and sequential logic optimization which reduces area and improves performance of the circuit, technology mapping which maps hardware resources onto architectural components, and placement and routing which optimizes the layout and connections between resources.

6.2 Synthesis Techniques for Binary Synthesis

Most binary synthesis approaches utilize existing high-level synthesis techniques which are made possible by initially decompiling to recover high-level constructs. Therefore, binary synthesis can apply any scheduling, resource allocation, binding, or control synthesis techniques. In addition, binary synthesis also uses existing low-level techniques for logic optimization, technology mapping, placement, and routing.

Dynamic binary synthesis approaches utilize specialized low-level synthesis techniques because existing methods require too much execution time and memory to run on-chip. On-chip logic minimization [Lysecky and Vahid 2003] techniques have been proposed that run up to 20x faster than existing approaches, use 3x less memory, while achieving results in some cases only 2% worse than existing techniques. Techniques have also been proposed for on-chip technology mapping, placement, and routing [Lysecky et al. 2004], resulting in speedups of 3x compared to existing approaches with an 18x reduction in memory. On-chip routing results in a 30% increase in the length of the critical path, which in many cases may be an acceptable trade-off to enable dynamic synthesis.

7. BINARY UPDATING

Binary updating techniques modify a software binary by adding or removing instructions or by changing existing instructions. Binary updating is a difficult problem because at the binary level, the compiler and linker have assigned all instruction addresses. Therefore, adding an instruction requires the binary updater to modify all control instructions to adjust their targets to the proper location. Also, if data is mixed with code in the binary, then the binary updater must also adjust all references to that data. Therefore, in order to add an instruction, the binary updater must consider the effects on all other instructions.

7.1 Summary of Binary Update Approaches

The problem of updating a binary was initially addressed by the tool EEL (Executable Editing Library) [Larus and Schnarr 1995] which is part of WARTS (Wisconsin Architectural Research Tool Set) [Hill et al. 1993]. EEL was originally intended for instrumenting an existing executable to perform profiling. EEL accomplished this instrumentation by adding additional instructions to keep track of execution frequencies. Other uses of EEL include emulation of features not provided in hardware and optimization of existing binaries. Etch [Romer et al. 1997] is a similar approach capable of rewriting Win32 x86

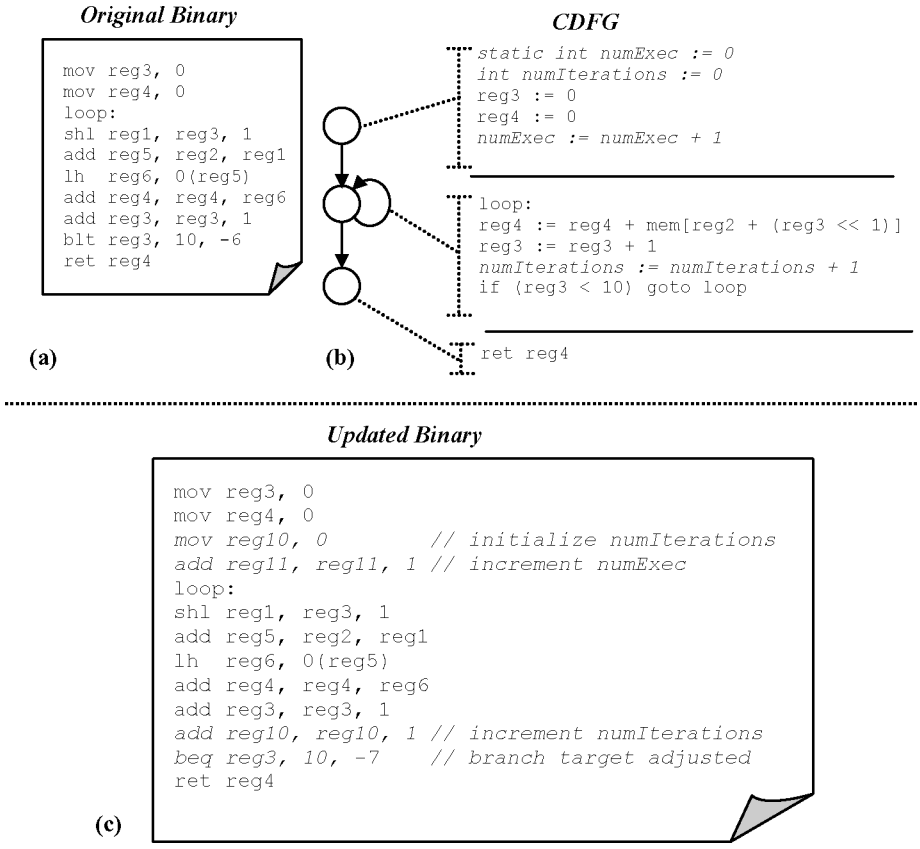

Fig. 4. Binary updating. (a) Original binary for an accumulate loop. (b) Initially, the binary updater converts the original binary to a control/data flow graph and then inserts new code (shown in italics), which in this example keeps track of the executions and iterations of the loop. (c) Next, the binary updater updates the binary by assembling the control/data flow graph. During assembly, the binary updater allocates registers for *numExec* and *numIterations* and adjusts all branch targets for the new instructions.

binaries. Pixie [MIPS 1990] and QPT [Ball and Larus 1994] are similar approaches. Dynamic binary synthesis approaches [Lysecky et al. 2006] utilize a simpler binary update approach that overwrites the instructions of a loop with code that initializes the hardware, and then jumps to the end of the original loop. The advantage of this approach is that the simplicity allows for dynamic binary updating. The disadvantage is that the instructions used to initialize the loop must be smaller than the instructions used to implement the loop.

7.2 Techniques for Updating Binaries

Figure 4 illustrates the binary update process. Figure 4(a) shows the original binary for an accumulate loop. The binary updater initially converts the binary into a control/data flow graph, shown in Figure 4(b) in order to remove the absolute instruction addresses. This conversion allows the binary updater to insert additional code without having to modify every other control instruction.

Figure 4(b) shows an example of inserted code (shown in italics) to profile the iterations and executions of the loop in the binary. This loop is either identified by decompilation applied during binary synthesis or by decompilation applied by the binary updater. When using the binary updater as part of binary synthesis, the binary updater uses the decompiled representation as input. Finally, the binary updater assembles and links the modified control/data flow graph, creating a new binary as shown in Figure 4(c). During assembly, the binary updater allocates registers for the inserted code if possible and reassigns the targets of the control instructions to the appropriate location. For the example in the figure, the binary updater stored *numIterations* in register reg10 and *numExec* in register reg11. The binary updater modified the target of the branch at the end of the original binary to account for the additional add instruction used to increment *numIterations*.

7.3 Uses of Binary Updating During Binary Synthesis

Binary synthesis utilizes binary updating for two purposes: instrumenting the binary with profiling code and inserting instructions to initialize and communicate with hardware. Using the binary updater to profile was illustrated in Figure 4 where the binary updater included additional instructions to monitor iterations and execution frequencies of a loop.

Binary updating implements communication with hardware by adding instructions to transfer all inputs and outputs to/from each hardware region. During decompilation, definition-use analysis identifies registers that represent inputs and outputs to a region. Definition-use analysis identifies a register as an input if that register is used in the region before it is defined. Similarly, definition-use analysis identifies a register as an output if that register is defined by the region and is used again outside the region before it is redefined. In addition to identifying register inputs, the decompiler also identifies array inputs and scalar inputs. The input for an array is the base address of the array and the amount of data that is read from the array, both of which are identified during the array recovery step of decompilation. Definition-use analysis is also used to identify scalar inputs. In the situation where an address of an array or scalar cannot be determined statically, the hardware reads the inputs from memory during execution after computing the appropriate address.

Figure 5 illustrates how binary updating modifies the binary to communicate with synthesized hardware during binary synthesis. Figure 5(a) shows the original binary for a function that consists of an accumulate loop. This binary has a single input, reg2 which stores the base address of the array to be accumulated. The binary also has a single output, reg4, which stores the accumulated value. The registers reg3, reg4, reg5, and reg6 can also be considered outputs because they are defined by the function. However, because the code does not use these registers again, we do not consider them as outputs. Figure 5(b) shows a simple synthesized circuit for the corresponding binary. The circuit consists of an adder to accumulate values in the array and a DMA to fetch the array from memory. The circuit has four register inputs: *Addr* (the base address of the

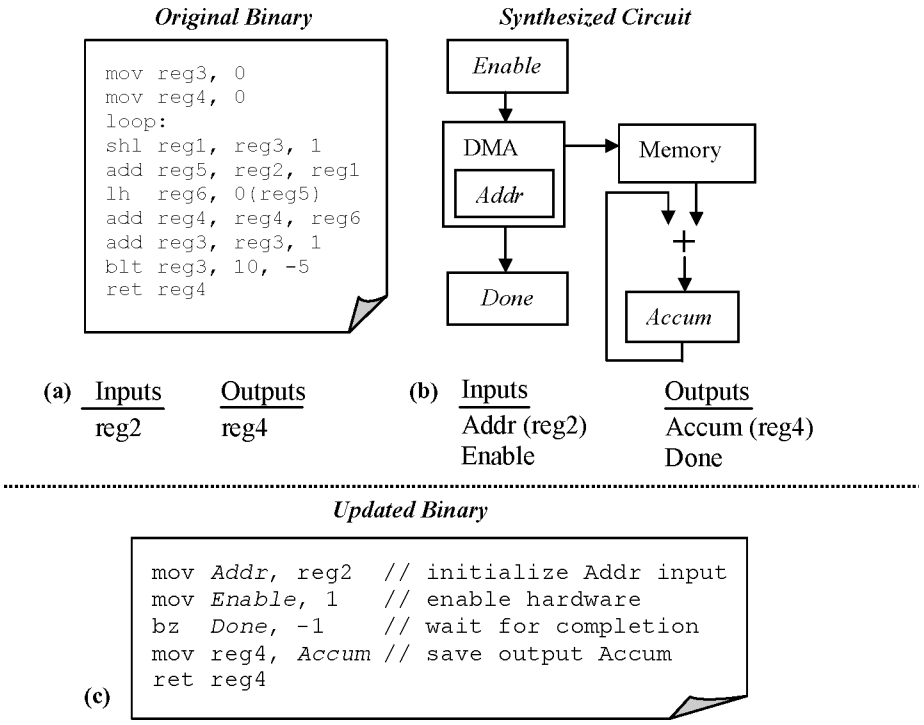

Fig. 5. Communication with hardware using binary updating. (a) The original binary for an accumulate loop, with hardware input reg2 (base address of the array) and output *reg4* (accumulated value). (b) A simple hardware circuit for the corresponding binary, with memory mapped registers for *Enable, Addr, Done*, and *Accum*. (c) An updated binary that uses the synthesized circuit.

array, specified by reg2), *Accum* (the accumulated value to be stored into reg4), *Enable*, and *Done*. Figure 5(c) shows the updated binary that communicates with the synthesized circuit, assuming for simplicity that the architecture supports memory-mapped hardware registers. Initially, the code moves reg2 into the *Addr* input register and then enables the hardware by moving the value 1 into the *Enable* input register. The following branch instruction loops until the *Done* register is set upon the completion of the DMA transfer, essentially stalling the microprocessor until the hardware finishes. Alternatively, the binary updater may use instructions to put the microprocessor to sleep to save power. Next, the binary updater uses a move instruction to move the accumulated value from the *Accum* output register into reg4 which gets returned from the function. If we had considered registers 3, 4, 5, and 6 as outputs, then the binary updater would also have included instructions for assigning these registers upon completion of the hardware. For this example, the target architecture uses memory-mapped registers within the FPGA, allowing a move instruction to transfer data to these registers. Depending on the instruction set, binary updating may use load instructions. Other communication models, such as a shared cache or shared memory with DMA, would require different instructions.

The execution time required by the added instructions varies depending on the architecture. For the memory-mapped register example, an access to a memory-mapped register by the microprocessor typically takes approximately 10 cycles. Therefore, the updated binary in Figure 5(c) may require an additional 40 cycles, one for each register access. Although binary updating may seem to create an overhead due to the addition of communication instructions, such instructions would also be added by high-level hardware/software partitioning tools.

8. EXISTING BINARY SYNTHESIS APPROACHES

We initially proposed binary synthesis in Stitt and Vahid [2002] to achieve a more transparent tool-flow integration for hardware/software partitioning compared to traditional partitioning approaches. This initial approach utilized limited decompilation to recover high-level control structures and to optimize away assembly inefficiencies, achieving speedups compared to a MIPS microprocessor at the cost of a large area overhead. The area overhead was later reduced using additional decompilation techniques which resulted in significant speedups compared to software execution for the standard benchmark suites MediaBench, NetBench, and EEMBC [Stitt and Vahid 2003].

Mittal et al. [2004] developed similar binary synthesis techniques to translate DSP binaries into FPGA hardware. By treating the binaries as an intermediate representation for synthesis, their approach could synthesize applications written in C/C++, Matlab, and Simulink in addition to hand-optimized assembly code. The FREEDOM Compiler [Zaretsky et al. 2004] expands on this approach by converting the binary to a machine syntax tree, optimizing the tree, creating a control/data flow graph, and then performing synthesis with resource constraints specified by an architectural description library.

In Stitt et al. [2005a], we showed that binary synthesis could take advantage of advanced synthesis techniques, such as smart buffers [Guo et al. 2004], by performing more decompilation to recover memory access patterns and arrays. This work also showed that decompiled C code commonly results in similar synthesized hardware compared to synthesis from the original C code. We also presented a case study of binary synthesis on a commercial h.264 decoder [Stitt et al. 2005b]. Whereas previous efforts typically showed the possibility of performing binary synthesis on small, unoptimized benchmarks, this work showed that binary synthesis could be competitive with high-level synthesis approaches on a large, highly-optimized commercial application. In Stitt and Vahid [2005b], we introduced loop rerolling and strength promotion as decompilation techniques that greatly improve the quality of hardware synthesized from a software binary in the presence of software compiler optimizations. Stitt and Vahid [2005b] also showed that binary synthesis can produce similar results for several different instruction sets, including instruction sets of softcore microprocessors such as the MicroBlaze.

Critical Blue [CriticalBlue 2006] is a commercial binary synthesis approach that utilizes decompilation to synthesize a custom VLIW coprocessor to

speed up critical software kernels. Binachip [2006] is a similar approach that translates a software binary into a custom FPGA implementation.

Several attempts at synthesizing Java byte code have been proposed [Helaihel and Olukotun 1997; Kuhn et al. 1999]. Java byte code is potentially a good candidate for binary synthesis because high-level information is stored in the byte code, such as class and thread information. Due to the difficulty in extracting parallelism from a sequential application, having explicit coarse-grained parallelism in the byte code makes synthesis of byte code potentially superior to synthesis from languages without explicit parallelism. The main difficulty in synthesizing Java byte code is dealing with the abundance of virtual functions which result in indirect jumps that make static control flow analysis difficult. In addition, the use of references requires complex alias analysis. Helaihel [Helaihel and Olukotun 1997] eliminates many of these problems by preallocating class instances, but this approach is limited to objects that are not created in loops or in recursive functions. Kuhn et al. [1999] restricts the constructs and coding style of Java code to make RTL, behavioral, and system synthesis possible.

In addition to static binary synthesis techniques, warp processors [Lysecky et al. 2006; Stitt et al. 2003] perform binary synthesis at runtime, providing a completely transparent synthesis tool flow. Warp processors include extra hardware to profile an executing binary to identify critical regions and also contain a specialized configurable logic fabric that supports efficient on-chip CAD tools that run on a small coprocessor. Beck and Carro [2005] proposed a similar, more coarse-grained approach that maps sequences of Java byte code onto a coarse-grained reconfigurable fabric, achieving a speedup of 4.6x compared to a stack-based java virtual machine.

9. CONCLUSIONS AND FUTURE WORK

We introduced binary synthesis as a complementary approach to traditional high-level synthesis techniques. Compared to high-level synthesis approaches, binary synthesis can be more transparently integrated into existing software tool flows by supporting all high-level languages and compilers. Binary synthesis also creates the possibility of synthesizing library code and legacy applications.

We showed that decompilation is an essential step in binary synthesis, recovering high-level information that is lost during software compilation. We presented the underlying techniques used by most decompilers and discussed the limitations of existing techniques. By using decompilation, binary synthesis approaches have achieved very competitive results compared to traditional synthesis techniques.

With the increasing popularity of new languages and compilers, binary synthesis is likely to become more widely accepted due to its independence from languages and compilers. Binary synthesis enables a platform provider to release a single binary synthesis tool for the instruction set used by the platform as opposed to supporting each possible language that a user of a particular platform might consider. By not excluding certain languages or compilers, binary

synthesis can also attract a new market segment of software developers to use microprocessor/FPGA platforms. Also, the increasing possibilities of dynamic binary synthesis allow software developers to obtain software speedups with no additional effort and without any changes to tool flow.

Future work will likely involve extending binary synthesis to handle parallel applications in which case more complex decompilation and partitioning techniques will be needed to recover threads, processes, and communication/synchronization constructs. In addition, the actual performances obtainable for different languages are currently unknown because most existing approaches only synthesize binaries generated from C code. Indirect jumps in C++ and Java code resulting from virtual functions can result in inefficient hardware. However, this problem is not unique to binary synthesis; it also applies to traditional synthesis approaches. Future work in dynamic binary synthesis will likely include improved synthesis and placement and routing algorithms that can execute extremely quickly while still resulting in fast hardware, making dynamic binary synthesis more competitive with static approaches. Such work may also include handling multithreaded and multiprocess binaries.

REFERENCES

AHPAH SOFTWARE INC. 2004. SourceAgain Java decompiler. http://www.ahpah.com/product.html.

ALTERA CORP. 2006. Nios embedded processor. http://www.altera.com/products/ip/processors/nios/nio-index.html.

AMMONS, G., BALL, T., AND LARUS, J. R. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 85–96.

ATHANAS, P. AND SILVERMAN, H. 1993. Processor reconfiguration through instruction-set metamorphosis. *IEEE Comput. 26*, 3, 11–18.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices 35*, 5, 1–12.

BALL, T. AND LARUS, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*, 4, 1319–1360.

BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 191–201.

BARBE, P. 1974. The Piler system of computer program translation. Techn. rep. PLR-020, Probe Consultants Inc. Prepared for the Office of Naval Research, distributed by National Technical Information Service, USA. ADA000294. Contract N00014-67-C-0472.

BECK, A. C. S. AND CARRO, L. 2005. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In *Proceedings of the Design Automation Conference (DAC)*. 732–737.

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement, and routing for FPGA research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications (FPLA)*. 213–222.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.

BINACHIP, INC. 2006. http://www.binachip.com/.

BÖHM, W., HAMMES, J., DRAPER, B., CHAWATHE, M., ROSS, C., RINKER, R., AND NAJJAR, W. 2002. Mapping a single assignment programming language to reconfigurable systems. *Supercomput. 21*, 117–130.

THE BOOMERANG DECOMPILER PROJECT. 2006. Boomerang Decompiler. http://boomerang.sourceforge.net/.

BRAYTON, R. K., HATCHTEL, G. D., MCMULLEN, C. T., AND SANGIOVANNI-VINCENTELLI, A. L. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers.

BREUER, P. T. AND BOWEN, J. P. 1994. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Sys. 16*, 5, 1613–1647.

BRINKLEY, D. L. 1981. Intercomputer transportation of assembly language software through decompilation. Techn. rep., Naval Underwater Systems Center.

CELOXICA. 2006. DK design suite. http://www.celoxica.com/products/dk/default.asp.

CHAN, J. T. AND YANG, W. 2004. Advanced obfuscation techniques for Java bytecode. *J. Syst. Softw. 71*, 1–2, 1–10.

CHERNOFF, A., HERDEG, M., HOOKWAY, R.,REEVE, C., RUBIN, N., TYE, T., BHARADWAJ YADAVALLI, S., AND YATES, J. 1998. FX!32 a profile-directed binary translator. *IEEE Micro 18*, 2, 56–64.

CHIODO, M., GIUSTO, P., JURECSKA, A., HSIEH, H., SANGIOVANNI-VINCENTELLI, A., AND LAVAGNO, L. 1994. Hardware-software codesign of embedded systems. *IEEE Micro 14*, 4, pp. 26–36.

CHOU, Y. AND SHEN, J. P. 2000. Instruction path coprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 270–281.

CIFUENTES, C. 1994. Reverse compilation techniques. PhD thesis Queensland University of Technology, Department of Computer Science.

CIFUENTES, C. AND VAN EMMERIK, M. 2000. UQBT: Adaptable binary translation at low cost. *IEEE Comput. 33*, 3, 60–66.

CIFUENTES, C., VAN EMMERIK, M., UNG, D., SIMON, D., AND WADDINGTON, T. 1999. Preliminary experiences with the use of the UQBT binary translation framework. In *Proceedings of the Workshop on Binary Translation*. 12–22.

CIFUENCES, C., WADDINGTON, T., AND VAN EMMERIK, M. 2001. Computer security analysis through decompilation and high level debugging. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 375–380.

CLARK, N., BLOME, B., CHU, M., MAHLKE, S., BILES, S., AND FLAUTNER, K. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 272–283.

CLARK, N., KUDLUR, M., PARK, H., MAHLKE, S., AND FLAUTNER, K. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 30–40.

CRITICALBLUE. 2006. http://www.criticalblue.com.

DECOMPILATION WIKI. 2006. http://www.program-transformation.org/Transform/DeCompilation.

DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. 2003. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 15–24.

DE MICHELI, G. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill.

DINIZ, P., HALL, M., PARK, J., SO, B., AND ZIEGLER, H. 2005. Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis systems. *J. Microprocess. Microsyst. 29*, 2–3, 51–62.

DISC DECOMPILER. 2006. http://www.debugmode.com/dcompile/disc.htm.

DUESTERWALD, E. AND BALA, V. 2000. Software profiling for hot path prediction: less is more. *SIGPLAN Notices 35*, 11, 202–211.

EBCIOGLU, K., ALTMAN, E., GSCHWIND, E., AND SATHAYE, S. 2001. Dynamic binary translation and optimization. *Trans. Comput. 50*, 6, 529–548.

ELES, P., PENG, Z., KUCHCHINSKI, K. AND DOBOLI, A. 1997. System level hardware/software partitioning based on simulated annealing and tabu search. *J. Design Autom. Embedd. Syst. 2*, 1, 5–32.

ENZLER, R., JEGER, T., COTTET, D., AND TROSTER, G. 2000. High-level area and performance estimation of hardware building blocks on FPGAs. In *Proceedings of the Roadmap to Reconfigurable Computing, Workshop on Field-Programmable Logic and Applications (FPL)*. Vol. 1896, 525–534.

ERNST, R. AND HENKEL, J. 1992. Hardware-software codesign of embedded controllers based on hardware extraction. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*.

FIN, A., FUMMI, F., AND SIGNORETTO, M. 2001. SystemC: a homogenous environment to test embedded systems. In *Proceedings of the International Workshop on Hardware/Software Codesign (CODES)*. 17–22.

FISHER, J. 1999. Customized instruction-sets for embedded processors. In *Proceedings of the Design Automation Conference (DAC)*. 253–257.

FLEURY, M., SELF, R. P., AND DOWNTON, A. C. 2001. Hardware compilation for software engineers: An ATM example. *Softw. Engin. 148*, 1, 31–42.

FRIEDMAN, F. L. AND SCHNEIDER, V. B. 1973. A systems implementation language. *SIGPLAN Notices 8*, 9, 60–63.

FRIENDLY, D. H., PATEL, S. J., AND PATT, Y. N. 1998. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 173–181.

FRIGO, J., GOKHALE, AND M., LAVENIER, D. 2001. Evaluation of the streams-C C-to-FPGA compiler: An applications perspective. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. 134–140.

FUAN, C. AND ZONGTIAN, L. 1991. C function recognition technique and its implementation in 8086 C decompiling system. *Mini-Micro Syst. 12*, 11, 33–40.

GAJSKI, D., DUTT, N., WU, A., LIN, S. 1992. *High-Level Synthesis—Introduction to Chip and System Design*. Kluwer Academic Publishers.

GAJSKI, D., VAHID, F., NARAYAN, S., AND GONG, J. 1994. *Specification and Design of Embedded Systems*. Prentice Hall.

GOKHALE, M. B. AND STONE, J. M. 1998. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *Proceedings of Symposium on FPGAs for Custom Computing Machines (FCCM)*. 126–135.

GONZALEZ, R. E. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro 20*, 2, 60–70.

GORDON-ROSS, A. AND VAHID, F. 2003. Frequent loop detection using non-intrusive on-chip hardware. In *Proceedings of International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*. 1203–1215.

GSCHWIND, M., ALTMAN, E., SATHAYE, S., LEDAK, P., AND APPENZELLER., D. 2000. Dynamic and transparent binary translation. *IEEE Comput. 33*, 3, 54–59.

GUO, Z., BUYUKKURT, A. B., AND NAJJAR, W. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 249–256.

GUPTA, R. K. AND DE MICHELI, G. 1993. Hardware-software cosynthesis for digital systems. *IEEE Design Test Comput. 10*, 3, 29–41.

GUPTA, R. K. AND DE MICHELI, G. 1992. System-level synthesis using re-programmable components. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2–7.

GUPTA, S., DUTT, N. D., GUPTA, R. K., AND NICOLAU, A. 2003. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of International Conference on VLSI Design (VLSI)*.

HALSTEAD, M. H. 1967. Machine independence and third generation computers. In *Proceedings of Sprint Joint Computer Conference (SJCC)*. 587–592.

HALSTEAD, M. H. 1970. Using the computer for program conversion. *Datamation*, 125–129.

HELAIHEL, R. AND OLUKOTUN, K. 1997. Java as a specification language for hardware-software systems. In *Proceedings of the International Conference on Computer-aided Design (ICCAD)*. 690–697.

HILL, M. D., LARUS, J. R., LEBECK, A. R., TALLURI, M., AND WOOD, D. A. 1993. Wisconsin architectural research tool set. *SIGARCH Computer Architecture News 21*, 4.

HOLLANDER, C. R. 1973. Decompilation of object programs. PhD dissertation, Computer Science, Stanford University.

HOOD, S. T. 1991. Decompiling with definite clause grammars. Tech. rep. ERL-0571-RR, Electronics Research Laboratory, DSTO Australia.

HOPWOOD, G. L. 1978. Decompilation. PhD dissertation, Computer Science, University of California, Irvine.

HOUSEL, B. C. AND HALSTEAD, M. H. 1974. A methodology for machine language decompilation. In *Proceedings of the ACM Annual Conference*, ACM Press, 254–260.

JONES, A. K., HOARE, R., KUSIC, D., FAZEKAS, J., AND FOSTER, J. 2005. An FPGA-based VLIW processor with custom hardware execution. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. 107–117.

JREVERSEPRO. 2006. http://jrevpro.sourceforge.net/.

KANNAN, P., BALACHANDRAN, S., AND BHATIA, D. 2002. On metrics for comparing routability estimation methods for FPGAs. In *Proceedings of the Design Automation Conference (DAC)*. 70–75.

KU, D. AND DE MICHELI, G. 1990. HardwareC—a language for hardware design (version 2.0). Tech. rep. CSL-TR-90-419, Stanford University.

KUCUKCAKAR, K. 1999. An ASIP design methodology for embedded systems. In *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*. 17–21.

KUHN, T., ROSENTIEL, W., AND KEBSCHULL, U. 1999. Description and simulation of hardware/software systems with Java. In *Proceedings of the Design Automation Conference (DAC)*. 790–793.

KULKARNI, D., NAJJAR, W., RINKER, R., AND KURDAHI, F. 2002. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. In *Proceedings of Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 239.

LARUS, J. R. AND SCHNARR, E. 1995. EEL: Machine-independent executable editing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 291–300.

LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., AND STOCKWOOD, J. 2000. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the Design Automation Conference (DAC)*. 507–512.

LYSECKY, R., STITT, G., AND VAHID, F. 2006. Warp processors. ACM *Trans. Design Autom. Electron. Syst. 11*, 3, 659–681.

LYSECKY, R. AND.VAHID, F. 2003. On-chip logic minimization. In *Proceedings of the Design Automation Conference (DAC)*. 334–337.

LYSECKY, R., VAHID, F., AND TAN, S. 2004. Dynamic FPGA routing for just-in-time compilation. In *Proceedings of the Design Automation Conference (DAC)*. 954–959.

MENTOR GRAPHICS CORP. 2006. Catapult C synthesis. http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm.

MIPS COMPUTER SYSTEMS. 1990. UMIPS-V reference manual (pixie and pixstats).

MITTAL, G., ZARETSKY, D., TANG, X., AND BANERJEE, P. 2004. Automatic translation of software binaries onto FPGAs. In *Proceedings of the Design Automation Conference (DAC)*. 389–394.

MOCHA, THE JAVA DECOMPILER. 1996. http://www.brouhaha.com/~eric/software/mocha/

MYCROFT, A. 2001. Comparing type-based and proof-directed decompilation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 362–367.

MYCROFT, A. 1999. Type-based decompilation. In *Proceedings of the European Conference on Programming (ESOP)*. 208–223.

NAJJAR, W., BÖHM, W., DRAPER, B., HAMMES, J., RINKER, R., BEVERIDGE, R., CHAWATHE, M. AND ROSS, C. 2003. From algorithms to hardware—A high-level language abstraction for reconfigurable computing. *IEEE Comput. 6*, 8, 63–69.

OXFORD HARDWARE COMPILATION GROUP. 1997. The Handel language. Tech. rep., Oxford University.

REC (REVERSE ENGINEERING COMPILER). 2005. http://www.backerstreet.com/rec/rec.htm.

ROMER, T., VOELKER, G., LEE, D., WOLMAN, A., WONG, W., LEVY, H., CHEN, B., AND BERSHAD, B. 1997. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*. 1–7.

SASSAMAN, W. A. 1966. A computer program to translate machine language into Fortran. In *Proceedings of SJCC*. 235–239.

SCHNEIDER, V. AND WINIGER, G. 1974. Translation grammars for compilation and decompilation. *BIT Numer. Mathem. 14*, 1, 78–86.

SCOTT, J., LEE, L.H., CHIN, A., ARENDS, J., AND MOYER, W. 1999. Designing the M*CORE M3 CPU architecture. In *Proceedings of the International Conference on Computer Design (ICCD). 94*.

SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. 1993. Binary translation. *Comm. ACM 36*, 2, 69–81.

SRINIVASAN, V., RADHAKRISHNAN, S., AND VEMURI, R. 1998. Hardware/software partitioning with integrated hardware design space exploration. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE)*. 28–35.

STITT, G., GUO, Z., VAHID, F., AND NAJJAR, W. 2005a. Techniques for synthesizing binaries to an advanced register/memory structure. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)*. 118–124.

STITT, G., LYSECKY, R., AND VAHID, F. 2003. Dynamic hardware/software partitioning: A first approach.In *Proceedings of the Design Automation Conference (DAC)*. 250–255.

STITT, G. AND VAHID, F. 2003. Binary-level hardware/software partitioning of MediaBench, NetBench, and EEMBC benchmarks. Tech. rep. UCR-CSE-03-01, Department of Computer Science and Engineering, University of California, Riverside.

STITT, G. AND VAHID, F. 2005a. A decompilation approach to partitioning software for microprocessor/FPGA platforms. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. 396–397.

STITT, G. AND VAHID, F. 2002. Hardware/software partitioning of software binaries. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. 164–170.

STITT, G. AND VAHID, F. 2005b. New decompilation techniques for binary-level co-processor generation. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. 547–554.

STITT, G., VAHID, F., MCGREGOR, G., AND EINLOTH, B. 2005b. Hardware/software partitioning of software binaries: A case study of H.264 decode. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*. 285–290.

STRETCH, INC. 2006. http://www.stretchinc.com.

SYNOPSYS. 2006. Design compiler. http://www.synopsys.com/products/logic/design_compiler.html.

SYNPLICITY, INC. 2006. Synplify ASIC. http://www.synplicity.com/corporate/pressreleases/2003/SYB-197final.html

TENSILICA, INC. 2006. XPRES Compiler. http://www.tensilica.com/html/xpres.html.

TRANSMETA, CORP. 2006. Transmeta Efficeon. http://www.transmeta.com/efficeon/.

VAHID, F. 1997. Modifying min-cut for hardware and software functional partitioning. In *Proceedings of the International Workshop on Hardware/Software Co-Design (CODES)*. 43–48.

VAHID, F. AND GAJSKI, D. 1992. Specification partitioning for system design. In *Proceedings of the Design Automation Conference (DAC)*. 219–224.

VASWANI, K., THAZHUTHAVEETIL, M. J., AND SRIKANT, Y. N. 2005. A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 217–228.

WALKER, R. AND CAMPOSANO, R. 1991. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers.

WOLF, W. 1994. Hardware-software co-design of embedded systems. *Proceedings of the IEEE 82*, 7, 967–989.

WORKMAN, D. A. 1978. Language design using decompilation. Tech. rep., University of Central Florida.

XILINX, INC. 2006. Xilinx MicroBlaze. http://www.xilinx.com/xlnx/xebiz/designResources/ip_product _details.jsp?key=micro_blaze.

XU, M. AND KURDAHI F. 1996. Accurate prediction of quality metrics for logic level designs targeted towards lookup table based FPGAs. *Trans. VLSI Syst. 7*, 4, 411–418.

ZARETSKY, D., MITTAL, G., TANG, X. AND BANERJEE, P. 2004. Overview of the FREEDOM compiler for mapping DSP software to FPGAs. In *Proceedings of the Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 37–46.

Received February 2006; revised June 2006; accepted October 2006