Platform-Aware Bottleneck Detection for Reconfigurable Computing Applications

SETH KOEHLER, GREG STITT, and ALAN D. GEORGE, NSF Center for High Performance Reconfigurable Computing, University of Florida

Reconfigurable Computing (RC) has the potential to provide substantial performance benefits and yet simultaneously consume less power than traditional microprocessors or GPUs. While experimental performance analysis of RC applications has previously been shown crucial for achieving this potential, existing methods still require application designers to manually locate bottlenecks and determine appropriate optimizations, typically requiring significant designer expertise and effort. Worse, the diversity of platforms employed by RC applications further complicates the process of detecting bottlenecks and formulating optimizations. To address these shortcomings, we first discuss our platform-template system, which enables a performance analysis tool to perform more accurate bottleneck detection and achieve a higher degree of portability across diverse FPGA systems. We then provide details for our implementation of these concepts and techniques in the Reconfigurable Computing Application Performance (ReCAP) tool. Next, we present a taxonomy of common RC bottlenecks, providing associated detection and optimization strategies for each bottleneck, which we use to populate ReCAP's knowledge base for bottleneck detection. Finally, we demonstrate the utility of our approach via two application case studies across a total of three platforms.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*

General Terms: Measurement, Performance

Additional Key Words and Phrases: FPGA, optimization, performance analysis

ACM Reference Format:
Koehler, S., Stitt, G., and George, A. D. 2011. Platform-aware bottleneck detection for reconfigurable computing applications. ACM Trans. Reconfig. Technol. Syst. 4, 3, Article 30 (August 2011), 28 pages.
DOI = 10.1145/2000832.2000842 http://doi.acm.org/10.1145/2000832.2000842

1. INTRODUCTION

While performance continues to increase in both High-Performance Embedded Computing (HPEC) and High-Performance Computing (HPC), the growing importance of power has caused programmers in both fields to rely increasingly on explicit parallelism and accelerators rather than on instruction-level parallelism or clock frequency increases [Barroso 2005; Laudon 2005; Crawford et al. 2008]. Due to these trends, Reconfigurable Computing (RC), which typically employs both CPUs and reconfigurable hardware such as FPGAs, has emerged as a viable field for providing substantial performance [Garcia et al. 2006; Tessier and Burleson 2001] while simultaneously

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Aldec, Altera, GiDEL, Nallatech, Xilinx, and XtremeData.
Authors' addresses: S. Koehler (corresponding author), G. Stitt, A. D. George, NSF Center for High-Performance Reconfigurable Computing (CHREC), Electrical and Computer Engineering Department, University of Florida, Gainesville, FL 32611; email: garfieldsk@gmail.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1936-7406/2011/08-ART30 $10.00
DOI 10.1145/2000832.2000842 http://doi.acm.org/10.1145/2000832.2000842

consuming little power in comparison to microprocessors or GPUs [Williams et al. 2011, 2008; Che et al. 2008].

Performance analysis for RC applications has been shown crucial for achieving this potential performance due to the complexity of RC applications and systems [Koehler et al. 2008; Curreri et al. 2010]. In addition, automatic bottleneck detection has been shown a promising area of research in traditional HPC, accelerating the optimization process as well as reducing expertise needed by a designer to achieve a higher-performance solution for a given implementation and system [Mohr and Wolf 2003; Jorba et al. 2008; Chung et al. 2008; Su et al. 2009; Truong and Fahringer 2002]. Due to the added complexity of RC applications and systems, bottleneck detection is even more critical in RC, and yet is currently lacking, forcing application designers to manually locate performance bottlenecks as well as strategize optimizations.

Unfortunately, providing bottleneck detection for RC applications incurs additional challenges not typically present in traditional HPC. For example, while attempts have been made to standardize some aspects of RC-system APIs, such as OpenFPGA's GenAPI [OpenFPGA 2010], the continued widespread diversity of both hardware and software APIs for RC systems complicates bottleneck detection, as there are no standardized constructs such as `MPI_Send` (a construct within the MPI library that sends data from the calling node to another node in a system) that provide hooks for recording relevant statistics such as measured bandwidth or bytes transferred. Another key challenge exists in defining, locating, reporting on, and suggesting remedies for application bottlenecks. Many approaches in traditional HPC are ill-suited for RC applications due to common assumptions that all system processing elements are relatively homogeneous, general purpose, and at roughly the same level in the system hierarchy (i.e., applications execute on CPUs that can perform general computation and communication, individually or in groups). In contrast, components within an RC application are almost certainly heterogeneous, may be designed solely for scatter communication or pipeline control (noncomputational components), exist in fairly deep and intricate hierarchies, and may perform an arbitrary amount of work in a single cycle, thus resisting conventional bottleneck detection and classification schemes. In addition, an FPGA's hardware flexibility significantly expands the possibilities for optimization, increasing the complexity of formulating bottleneck remedies. Finally, from a practical standpoint, trace data, which is used heavily for bottleneck detection in traditional HPC, cannot be relied on in FPGAs as these devices typically have limited memories and real-time requirements. Specifically, it can be difficult to "pause" an application on an FPGA due to low-level interaction with external hardware, such as memories or other FPGAs.

Thus, in this article, we propose a platform-aware, knowledge-based bottleneck-detection framework to address the current lack of bottleneck detection for RC applications. The proposed framework and tool are extensible in that users may easily add support for their own platform, as long as it fits within the proposed platform-template model, and may also easily modify the bottleneck knowledge base. We also present what we believe to be the first taxonomy of common RC bottlenecks, including detection and optimization strategies, which we use to populate our ReCAP tool's knowledge base. Although this work should aid novice RC designers who may be unaware of many potential bottlenecks and optimizations, experienced RC designers can benefit as well from quick feedback on the location and severity of performance problems.

It is important to note that while we focus on improving runtime performance throughout this article, these concepts and techniques can also be of use for reducing power consumption or resource usage as well. For example, an increase in raw application performance may allow an application designer to decrease the number of hardware cores or the clock frequency on the FPGA while still providing the required performance. Conversely, optimization suggestions may indicate that resources could

be reduced without sacrificing performance, allowing these resources to be repurposed elsewhere for improved performance or to be left unused, reducing power or even permitting a smaller FPGA to be employed. Thus, optimizations may be used to achieve a balance amongst runtime performance, resource usage, and power consumption.

The remainder of this article is structured as follows. We first provide background and related research for performance analysis and automatic bottleneck detection in Section 2. Section 3 then details our platform-template system, enabling both platform-specific bottleneck detection and tool portability. Next, in Section 4, we describe our methodology for bottleneck detection and the nature of results produced by an extension to the Reconfigurable Computing Application Performance (ReCAP) tool. Then, in Section 5, we explore and categorize common performance bottlenecks in RC applications, including methods of detection as well as suggestions to remedy these bottlenecks. We then present case studies involving a time-domain finite impulse response benchmark and a two-dimensional probability density function estimator in Section 6, focusing on the utility of automatic bottleneck detection in the optimization process. Finally, we conclude and provide directions for further research in Section 7.

2. BACKGROUND AND RELATED RESEARCH

RC applications are typically programmed using both High-Level Languages (HLLs) for CPUs, such as C/C++, and Hardware Description Languages (HDLs) for FPGAs, such as VHDL or Verilog; while HLLs can also be used to describe hardware for an FPGA, we restrict our discussion to HDL-based RC applications. Although a microprocessor is optional, many RC applications use CPUs for tasks such as data movement and staging, pre- and postprocessing, or other computation that is better suited to a general-purpose processor. At runtime, CPUs and FPGAs communicate through one or more communication channels (e.g., PCIe, HyperTransport) in a variety of ways from HLL code including function/macro calls, class methods, and memory-mapped I/O, depending on the platform's API.

As stated in the Introduction, performance analysis, and specifically platform-aware, knowledge-based bottleneck detection, is critical for productively optimizing RC applications. Thus, in the remainder of this section we present background and related research for performance analysis (Section 2.1) and for platform-aware, knowledge-based bottleneck detection (Section 2.2).

2.1. Performance Analysis

In general, the goal of performance analysis is to aid the application designer in quickly locating and remedying performance bottlenecks, where a *bottleneck* refers to some portion of the application that reduces performance for the application as a whole (taken from the notion of a bottle's neck that restricts the flow of liquid from the larger bottle). As depicted in Figure 1, performance analysis may be divided into stages including gaining access to application data (instrumentation), recording and storing that data at runtime (measurement), optionally analyzing recorded data for performance problems (automated analysis), visualizing performance data and analysis results (presentation) in order to allow the designer to carry on further analyses (manual analysis), and finally strategizing and implementing changes within the application in order to ameliorate located performance bottlenecks (optimization). These steps may be repeated until desired performance is achieved or no further performance gains seem likely. It is important to note that application performance and potential bottlenecks may vary significantly based on the size and values of input data. Thus, it is critical to create performance tests that accurately reflect the gamut of expected inputs when optimizing. While instrumentation, measurement, and presentation significantly accelerate the process of optimizing an application, automated analysis and optimization are of

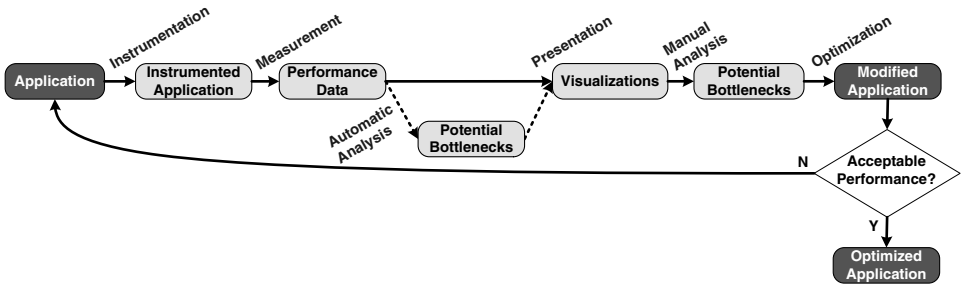


Fig. 1. Overview of performance analysis process.

increasing importance as applications, and thus recorded performance data, grow in complexity and size [Su et al. 2009; Chung et al. 2008; Mohr and Wolf 2003].

Our previous work provides performance analysis for RC applications, including instrumentation, measurement, and presentation for systems containing multiple CPUs and FPGAs in the ReCAP framework and tool, including both HDL-based and HLL-based applications [Koehler et al. 2008; Koehler and George 2010; Curreri et al. 2010]; to our knowledge, no other work provides general-purpose performance analysis for RC applications. As this work extends ReCAP for HDL-based applications, we provide a brief overview here. ReCAP builds upon the Parallel Performance Wizard (PPW) [Su et al. 2011], adding support for monitoring FPGA API calls and time spent in HDL conditional branches. Software may be written using C/C++, MPI, UPC, and SHMEM, while hardware may be written in VHDL, although significant work has been performed towards supporting Verilog.

The user's HDL code is first instrumented via the HDL Instrumenter (a Java GUI), which provides a number of options for profiling, tracing, and even verification (including assertion monitoring, code coverage, and HDL testbench generation for comparing actual execution with simulation). The HDL Instrumenter extracts and monitors all useful hardware signals in the FPGA, providing a mechanism to retrieve monitored FPGA data from software at runtime. Once instrumented, the user then employs their standard tool flow to synthesize and implement their HDL code, producing an instrumented FPGA configuration file used to program the FPGA. This configuration file is then added to a TAR file that was also produced during hardware instrumentation. The TAR file is then placed in the root software application folder, and the user's software code is instrumented and compiled using wrapper scripts (e.g., `ppwcc` instead of `cc`). Finally, the application is executed normally, producing performance data that may then be visualized in PPW's GUI or via SVG files produced from Graphviz. Figure 2 provides on overview of the process for analyzing an application's performance using ReCAP.

2.2. Knowledge-Based Bottleneck Detection and Platform Support

Knowledge-based bottleneck detection is a form of automated analysis designed to locate and describe common performance bottlenecks in an application based upon specific knowledge about where and how bottlenecks may occur. The goal of knowledge-based bottleneck detection is to reduce both the effort and expertise required to optimize an application, accelerating the optimization process; for a thorough overview of various frameworks and tools for automatic analysis; see Su et al. [2009]. Without bottleneck detection, the designer must understand the intricacies involving where bottlenecks can occur, understand how to detect each bottleneck, instruct the tool to monitor relevant data, interpret the performance data to locate bottlenecks, understand

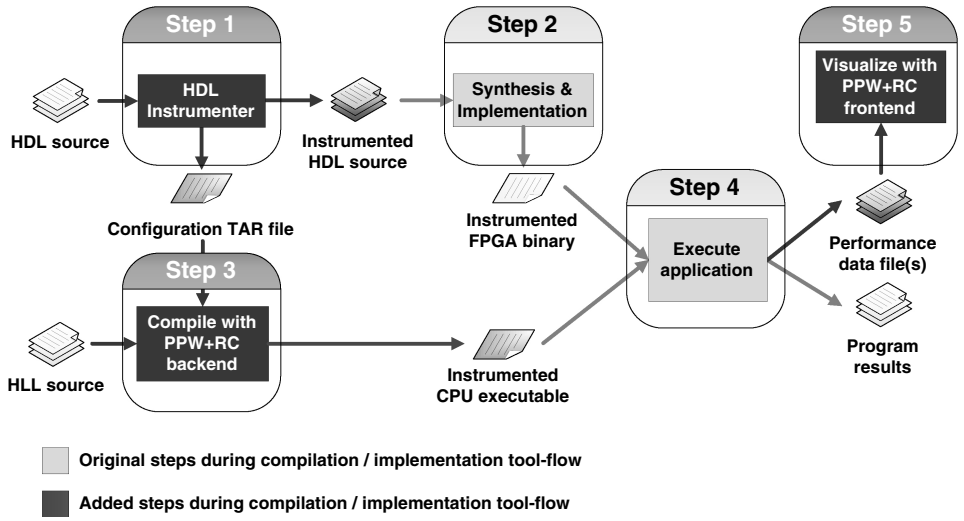


Fig. 2. Overview of instrumentation, measurement, and presentation in ReCAP.

what optimizations may be effective against each bottleneck, and determine the expected performance improvement if a given bottleneck were remedied (in order to ascertain whether the bottleneck is worth remedying).

To our knowledge, no research has been performed that enumerates or categorizes bottlenecks for RC applications, nor has automatic detection of RC bottlenecks been demonstrated. However, DeHon et al. [2004] present numerous "design patterns" for the purpose of improving application efficiency on reconfigurable systems; these design patterns could be included as optimization suggestions for relevant bottlenecks. For traditional HPC systems (i.e., systems without FPGAs), Mohr and Wolf [2003] provide a tree-based categorization of bottlenecks in EXPERT, a part of the KOJAK performance tool (now called Scalasca) that analyzes where an application spends time with respect to their categorization scheme. Jorba et al. [2008] discuss another knowledge-based bottleneck detection tool, KappaPI 2, that employs an XML format to store their knowledge base of bottleneck definitions, allowing the tool to locate patterns representing performance problems in trace data collected from message-passing applications. Static code analysis is used in conjunction with a specification of bottleneck "instances" (also in XML format) to indicate the cause or conditions of each bottleneck and to provide hints as to how the bottleneck may be remedied. Su et al. [2009] provide details of their automated analysis framework within Parallel Performance Wizard (PPW). Their tool provides automated, model-independent detection as well as distributed analysis capabilities to reduce analysis time. Truong and Fahringer [2002] provide a very detailed scheme for performance overhead classification in SCALEA, a profile- and trace-based performance analysis tool that allows user specification of metrics and code regions of interest. Chung et al. [2008] detail their framework and tool for automatic bottleneck detection using an extensible, rule-based system and performance data obtained from the IBM High-Performance Computing Toolkit. "Rules" are created using "metrics," both of which may be user-defined, and which, in turn, can depend on parameters such as the target system, enabling the tool to provide detailed information for all bottlenecks detected including expected performance improvement if the bottleneck was removed. However, as mentioned in the Introduction, there are a

number of differences in RC systems and applications that must be addressed for RC bottleneck detection.

Finally, due to the lack of standardized APIs for RC systems, tool designers must either attempt to support a myriad of platform APIs or provide the user with a mechanism to add support for their own platform. Due to the difficulties in adding and maintaining support for, or even gaining access to, each new platform and API version, the first approach will typically ensure a limited number of supported platforms and thus limited tool applicability. The latter approach is taken by some HLS tools, such as ImpulseC's platform-support packages [Bodenner 2010] or ROCCC's platform interface abstraction layer [University of California at Riverside 2010], as well as by frameworks supporting generic communication between heterogeneous devices, such as Auto-Pipe [Chamberlain et al. 2010] or the System-level Coordination Framework (SCF) [Aggarwal et al. 2009]. Unfortunately, since locating bottlenecks often requires additional information beyond what is necessary for portability (e.g., bytes transferred or expected bandwidth for each transfer type), such approaches are not well-suited for bottleneck detection. In addition, adding platform support may require significant expertise and effort (e.g., both ImpulseC and ROCCC require a manual conversion between a platform's API and their specified interface), limiting the potential for widespread use.

3. PLATFORM TEMPLATES

Platform templates have two purposes in ReCAP: bottleneck detection and portability. In this section we detail our platform-template framework and its implementation in ReCAP. While this framework handles many common API styles in both software and hardware, we will also point out cases not currently handled as well as concepts that could potentially address these situations.

ReCAP provides a single location within the HDL Instrumenter that encapsulates all platform-specific information, permitting a typical user to quickly add support for their own platform; this information is divided into several software and hardware API tabs for easier access. ReCAP allows each platform to be given a unique name and saves all platform information in a separate file, allowing platform templates to be easily loaded and shared. In fact, ReCAP can support user APIs built on top of a platform's API, allowing several different platform templates to coexist for the same platform; the user simply selects the corresponding template that matches the current API in use.

3.1. Software

In order to locate bottlenecks involving CPU-FPGA communication, or to even determine when such communication is occurring, ReCAP must know what API calls are possible as well as some auxiliary information such as purpose or bytes transfered (if a transfer) in order to effectively instrument these calls. Specifically, a user must enter a C/C++ prototype for a function, macro, or class method into the HDL Instrumenter[1]. As this information can be directly obtained from the platform API's header, which must be available on the system, the user can simply copy and paste this information into ReCAP.

For each prototype supplied, the user must indicate its type, which includes categories such as configuration, acquisition, data transfer, and release. In addition, C/C++ expressions can be given to identify the FPGA number associated with this call (if any), bytes transferred for data transfers, and the filename(s) used to configure the

[1]Macros are currently handled by providing an equivalent function prototype; while this approach could cause problems by presupposing argument and return types, an improved implementation could avoid this issue by handling macros separately.

FPGA. These expressions are free to access function arguments, global variables, class methods, or class public fields; for example, the FPGA number expression may simply be `fpgaNum` or `this->getFpgaId()`, assuming these represent an argument in the prototype or a public class method, respectively.

From this information, ReCAP identifies and instruments each API call in the user's source code. Function- and macro-based API calls are overridden via macros that effectively change the name of each API call in user source code in order to instead call instrumented wrapper functions or macros, which record various applicable statistics (e.g., time spent, bytes transferred, bandwidth achieved) in addition to performing the original API call[2]. Class methods are instrumented by subclassing each class within a platform API, with each subclass method recording statistics before calling the corresponding base class' method; in addition, any instantiation of a platform API class in the user's source code is replaced with an instantiation to the instrumented subclass in a copy of the user's source code. Unfortunately, this approach would not be sufficient for constructors and destructors due to the order in which constructors and destructors are called. For example, timing a class constructor in a platform's API would require a timer to be started before that constructor was called, whereas the subclass' constructor won't be called until after the platform API's constructor. To handle this situation, ReCAP's subclass employs multiple inheritance, first inheriting from another ReCAP-generated class that, in the case of the constructor, handles starting timers and other measurement code before the platform API's constructor is called; the subclass' constructor then stops timers and records statistics. Destructors are handled in the reverse fashion due to the reverse order in which destructors are called.

For bottleneck detection, ReCAP also allows a user to associate a default "reason" for an API call. Example "reasons" include initializing, broadcasting, or waiting to send due to a full buffer, and will be discussed in Section 4. For transfer functions, microbenchmark data can be entered in tabular form, where each row includes the transfer size (in bytes) and time taken to transfer that amount of data (in seconds). Microbenchmark data permits ReCAP to detect bottlenecks and make specific suggestions for communication-related bottlenecks, which are discussed in Section 5.1. ReCAP could also perform microbenchmarks automatically, such as upon installation or during a special calibration step, although additional information about how to use each API call would then be required. In the absence of microbenchmark data, ReCAP could simply detect performance deviations between different instances of the same API call. All of these techniques have been demonstrated in traditional HPC. Finally, a generic text field for platform-specific bottleneck suggestions is provided to associate known issues with given API calls. For example, on a Cray XD1 system [Cray 2010], it is roughly 200 times more efficient to have an FPGA write data into the CPU's memory than to have the CPU read from the FPGA's QDR SRAMs [Tripp et al. 2005].

A single API call could represent more than one behavior. For example, a transfer function may represent a read or a write depending on a class field or function argument, or standard library functions may be used to access the FPGA, such as an XtremeData XD1000 system's [XtremeData Inc. 2010] use of the standard C `open` function to initialize the FPGA. To support these scenarios, ReCAP permits a user to enter a condition to determine whether or not monitoring is active for a given API call, allowing a single transfer that can read and write to be monitored separately for each case; the user simply enters the prototype twice with conditions testing for a read

[2]The use of macros for overriding functions causes problems with C++'s function overloading, although method overloading is handled correctly; an improved implementation could handle this case properly via name-mangling for overloaded functions.

and write, respectively. This condition is equally useful for ignoring uses of standard functions (such as `open`) unless the correct arguments are given that indicate an FPGA access. A subtype label is also provided to allow a user to easily distinguish different conditional versions of the same API call in visualizations and bottleneck reports.

While the information discussed before is sufficient for monitoring and bottleneck detection, ReCAP must also be able to initiate transfers in order to retrieve hardware performance data at runtime, requiring additional information about how to actually perform these transfers. While a general implementation could provide additional detail about all transfer types, we reduce the amount of data a user must enter by requiring this additional information for only one send and receive type. Thus, the user must provide include directives for all FPGA libraries, a minimum and maximum transfer size permitted, the data type for FPGA transfers, and a short code fragment, usually one or two lines, that shows how to send (or receive) an array of data to (or from) the FPGA. ReCAP must also be aware of how to allocate, free, and access these arrays; default code for these tasks is provided, but can be overridden since FPGAs sometimes require page-aligned data, and thus special allocation functions and data structures. Further, ReCAP may be provided with conditions indicating a send or receive error for better error checking. Additional miscellaneous information can be provided as well, including a size multiplier on the FPGA data type for platforms that have different data widths in hardware and software, a file exclusion list to prevent instrumenting the inside of a user-defined API, and application-specific constants needed for FPGA access.

Unfortunately, some issues remain with our approach. First, some APIs employ memory-mapped FPGA access, where reading or writing to a specific memory location from the CPU actually constitutes an FPGA transfer. It is not possible, in general, to statically determine whether a given pointer access is in the FPGA's memory range, although runtime support is possible and many common cases could be detected statically. Thus, ReCAP currently requires wrapping such pointer accesses in simple macros or functions before they can be monitored; functions can be inlined to prevent performance loss. Second, we assume the read and write functions used by ReCAP to retrieve hardware performance data have addresses associated with them; in fact, during instrumentation, the user must provide an address range that is unused by the application so that ReCAP can hijack and use this range for transferring data. This address-based approach precludes hijacking stream-based API calls that lack address information; this issue could be resolved in a number of ways including hijacking another API call that does provide address information (since ReCAP only needs one address-based read and write API call), by embedding a special marker in the data stream and escaping that marker in any data sent by the application, or by setting a specific flag on the FPGA if such a feature were available and unused by the application. Despite these limitations, we have found that most platforms can be supported quickly (e.g., in a few hours). For example, ReCAP currently supports software and hardware APIs for a Nallatech PCI-X card [Nallatech 2010], an XtremeData XD1000 system [XtremeData Inc. 2010], and a GiDEL PROCStar III PCIe card [GiDEL 2010]; Table I in Section 6 contains more information on these platforms.

3.2. Hardware

In order to determine when communication occurs in hardware, ReCAP must know what events and data are associated with a read or write. Thus, ReCAP requests information concerning the names of data and address signals for both incoming and outgoing transfers as well as HDL conditions that indicate when data is available for, or when data can be sent by, the user's application. In addition, a user can provide an HDL code fragment to perform required actions when sending or receiving data,

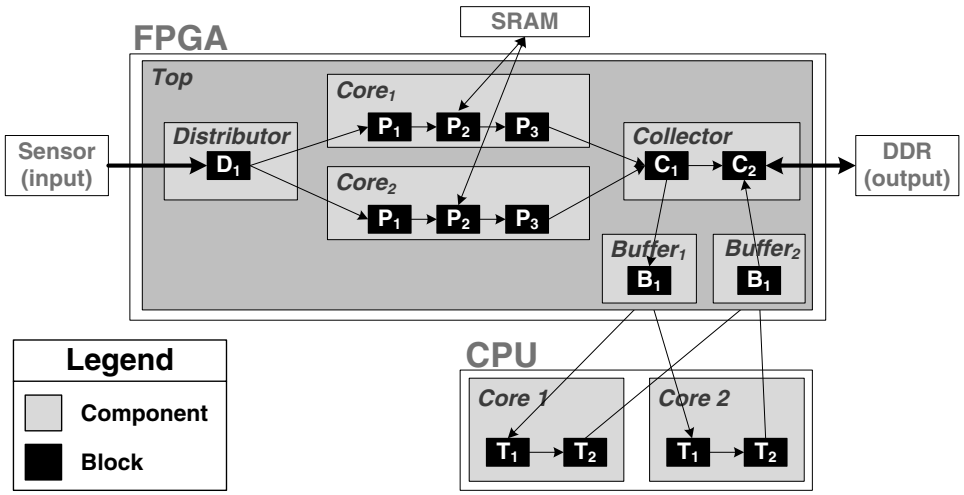


Fig. 3. Directed graph of an RC application that takes input from a sensor, processes data using a two-core pipeline, potentially offloads data to threads on a multicore CPU for further processing, and finally stores results in DDR memory.

such as by setting a valid or acknowledge flag high. Also, due to the wide variety of ways data can be transferred, ReCAP supports both memory-style access and block-transfer-style access. The memory-style access typically consists of an address, data, and a read/write enable; the block-transfer-style access (e.g., a DMA transfer) instead employs an address and the number of words to be transferred, a request flag, and potentially other handshaking flags as well. For block transfers, the user must provide the signal name indicating the number of words and the condition indicating a transfer request (address signals were already specified with the data signals earlier); the transfer ends when the number of words reaches 0. The user may also select whether the API will keep the address up-to-date on each cycle, or whether only a starting address is provided, in which case ReCAP will manage updating the address internally. Further, due to different latencies required by different platforms, the user may specify the number of cycles to delay outgoing data, including 0 for the same cycle. Finally, ReCAP requires the user to provide the top-level clock signal name (ReCAP currently only operates in one clock domain, although an extended implementation could support multiple clock domains) and reset condition.

As with software, we again assume an address-based scheme; if one is not present, techniques mentioned earlier are applicable here as well. We also note that our current framework only considers attachment to a CPU-FPGA communication port, whereas the FPGA may connect to another FPGA or external memory as well. We leave the extension of this framework to monitor these types of ports for future work, but this extension should be similar to the techniques presented here. Thankfully, this limitation only prevents *automatic* monitoring and bottleneck detection on these ports, as our current implementation can monitor any port via a few hardware pragmas in the top-level file; pragmas are discussed in Section 4.

4. BOTTLENECK DETECTION IN RC APPLICATIONS

As discussed in the Introduction, there are a number of factors beyond system and API diversity that complicate bottleneck detection in RC applications, including intricate hierarchies of interconnected components, component heterogeneity, and noncomputational components. For example, Figure 3 depicts a simple mockup RC application

that distributes sensor input between two application core pipelines, performs some computation using SRAM in that pipeline, collects results, and potentially offloads data for further processing to a dual-core CPU with two software threads per core before storing results in DDR memory. This application exhibits heterogeneity since many different blocks exist (e.g., the P1, P2, and P3 pipeline stages are likely performing fairly different tasks), there are noncomputational components such as buffers, distributors, and collectors, and there is complex interaction amongst blocks (e.g., data may traverse through 6 or 10 blocks via several paths). Thus, these differences must be addressed when detecting bottlenecks in RC applications.

We leverage our previous work in performance visualization and exploration [Koehler and George 2010] for abstracting application behavior as a directed graph of *blocks*, where a block represents a software thread, a clocked[3] VHDL "process" block, or a clocked Verilog "always" block (shown as black boxes in Figure 3). One key requirement of a block is that it operates in parallel with all other blocks, possibly with some dependencies due to interactions between blocks. VHDL "entity-architecture" pairs, Verilog "modules," and CPU cores are called *components*, and may contain one or more blocks. While each block may perform an arbitrary amount of communication and computation (e.g., a single block may be simultaneously communicating with several blocks while computing a multiply-and-accumulate), we choose a block to be the fundamental unit of parallelism in our abstraction of an application. This choice represents a tradeoff between a desire for detailed recording and modeling of fine-grained parallelism and a desire to minimize extraneous detail recorded and visualized.

Our previous work in performance visualization and exploration [Koehler and George 2010] also defines a pragma-based syntax, providing an application designer with a simple, unobtrusive methodology for specifying high-level information concerning application behavior; we extend this syntax here for the purpose of bottleneck detection. Figure 4 provides some examples of our extended syntax for software and hardware pragmas. Extensions to the syntax defined in our previous work include subdividing a "busy" category into "work" and internal "overhead" as well as the addition of a "reason" argument; for convenience, we briefly describe each part of a pragma's syntax in the following, including aspects defined in our previous work as well as these extensions.

Each pragma in software or hardware defines a *state* that the given block is in when that pragma is reached in source code; these pragmas are placed either before an API call or before the first statement in an HDL branch. Pragmas can indicate a block is working, performing internal overhead, communicating, or waiting. Specifically, the following categories are permitted.

—*Work*: tasks directly associated with the objective of the application (e.g., matrix multiplication or convolution); this should be maximized.
—*Overhead*: internal, auxiliary tasks that are artifacts of the implementation, such as bookkeeping or loop counter updates; this should be minimized.
—*Send*/*Recv*: movement of application data to or from this block.
—*Wait_send*/*Wait_recv*: synchronization with another block; examples include waiting to send data to a locked resource or waiting to receive data from a block that is currently working.

Additionally, a condition can be provided to indicate when a pragma is active; thus multiple pragmas can be defined per API call or HDL branch with conditions indicating which pragma is active at any given time. For example, only one of the two hardware pragmas in the "waitAck" state in Figure 4(b) is active, depending on whether the "ack"

[3]Purely asynchronous logic will already have had timing optimized during synthesis and implementation and can be monitored on clock boundaries if necessary.

```
#pragma recap writeX send(data, top.in, x1): words > 0
fpgaWrite(fpga, data, addr, words);
...
do {
  #pragma recap waitResult wait_recv(buffer_empty, top.out, r1)
  done = fpgaReadReg(fpga, addr2);
} while (done == 0);
```

(a) SW pragmas

```
case current_state is
when recvXCoord =>
--pragma recap getX recv(data, $CPU, x1)
...
when compute =>
--pragma recap mult work()
...
when nextIter =>
--pragma recap loopIncr overhead(update)
...
when waitAck =>
--pragma recap waitNext wait_recv(ack, top.out, w1): ack = '0'
--pragma recap next recv(ctrl, top.out, w1): ack = '1'
```

(b) HW pragmas

Fig. 4. Examples of user-defined pragmas.

signal is 0 or 1. All categories except for "work," which has no arguments, accept an optional first argument representing the "reason" the application is performing the given operation; we will provide exact details for "reasons" later in this section. Finally, the communication and wait categories require a *target* and *message ID* to be provided as the second and third arguments, respectively. A target indicates which block(s) the current block is communicating with or waiting for while a message ID distinguishes between different communication involving the same blocks.

Figure 4(a) shows several examples of software pragmas for bottleneck detection. The first pragma is given a unique, user-defined name of "writeX" and is categorized as a "send." The reason given is "data," indicating the following API call is sending data, as opposed to control messages, to a process block labeled "in" within the "top" VHDL entity/architecture. Note that targets are given using hierarchical references (such as is the case here) or using specially named blocks such as $CPU or $MEM. A message ID of "x1" is also given. Note that the condition indicates this pragma will be active only when "words" is greater than 0, as presumably the following API call does not represent communication if 0 words are sent. The second pragma, named "waitResult," indicates the following API call is waiting to receive data because of an empty buffer; the API call is waiting to receive data which will have a message ID of "r1" from the "top.out" process block.

Figure 4(b) shows several examples of hardware pragmas. The first, named "getX," is complementary to the first software pragma. The process block is described to be receiving data from the CPU with the same message ID. Note that it is possible to receive multiple message IDs if needed, such as when different threads or different states within the same thread interact with the same HDL pragma state. The next two pragmas indicate when the given block is working (e.g., performing some multiplication) and performing an internal overhead task with the "update" reason (e.g., updating a loop counter). The last two hardware pragmas demonstrate the use of conditions to differentiate behavior, even within the same state; in fact, no state machine needs

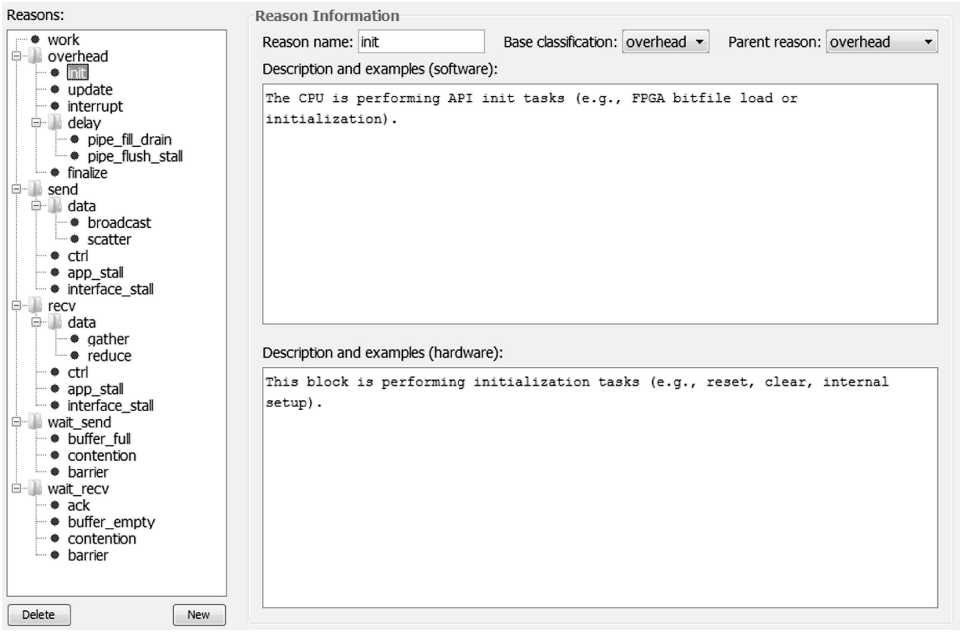


Fig. 5. Default "reasons" provided by ReCAP for classifying API calls or HDL branches.

to be present in hardware at all. The pragma named "waitNext" indicates the VHDL process block is waiting to receive an acknowledgment from another VHDL process block, "top.out," when the "ack" signal is low. However, when the "ack" signal is high, the second pragma, named "next," is active. This pragma indicates a control message, the acknowledge, is being received with the given message ID.

Bottlenecks can occur when one or more *blocks* are working slower than other blocks they interact with. In order to detect bottlenecks, we monitor time spent in each state defined by a pragma as well as transitions between these states, thus generating a Markov model of each block's behavior. ReCAP employs three constructs in bottleneck detection: *reasons*, *metrics*, and *bottleneck rules*; the latter two are discussed shortly and are similar in notion to the rules, metrics, and parameters in Chung et al. [2008] as well as to other techniques used in knowledge-based bottleneck tools [Jorba et al. 2008; Truong and Fahringer 2002].

Reasons provide an explanation for why an application is performing a given task. In addition to just indicating that a given block is waiting to receive data, a user can now specify the block is waiting to receive because of an empty buffer, because of contention, or both – multiple "reasons" can be provided for a single state. These "reasons" are user-editable and are closely tied to bottleneck rules discussed soon. For example, a contention bottleneck can be detected by searching for blocks that could achieve at least 1.05x speedup if all time spent in states with "contention" as a given "reason" were eliminated. Figure 5 shows the different "reasons" provided in ReCAP for each category.

The key concept behind "reasons" is that it is relatively easy for an application designer, if provided a list of "reasons," to look at an API call or branch of HDL code and determine whether it is waiting for an acknowledge, sending control messages, or updating a loop counter, whereas it is relatively difficult for a tool to ascertain such high-level information automatically. Conversely, it is possible for a tool, given

the preceding information for each API call or HDL branch of interest, to analyze the performance of each block for potential bottlenecks and account for block dependencies, whereas this task is fairly difficult for an application designer to perform manually.

Metrics are tool-provided measurements of runtime behavior such as duration of an event, bytes transferred, or bandwidth observed. Metrics can be reported for any block or overall, can be filtered by any combination of "reasons," and are generally used to define bottleneck rules, as discussed shortly. For example, a metric could return the minimum bandwidth for a software API function that was waiting to send due to a full buffer or waiting to receive due to an empty buffer, or a metric could return the total time a hardware block spent communicating or waiting. Metrics can be hardware- or software-specific, and currently include items such as total time spent; min/avg/max/total bytes transferred; min/avg/max bandwidth observed; number of calls, call groups, min/avg/max consecutive calls, and call type for software API calls; various statistics for microbenchmark data for a given API call, which are useful for comparing with actual bandwidth achieved; and miscellaneous metrics for computing formulas such as percentages and speedup. Unfortunately, adding, modifying, or removing metrics requires some detailed tool knowledge, and thus metrics are not user-customizable, although we have localized where metric information is defined to facilitate the inclusion of additional metrics.

Since it is desirable to have a more flexible, user-defined metric structure, we have begun development of a hardware directive framework for adding user-defined hardware modules (with some constraints on port interfaces) that could extend ReCAP's measurement capabilities, which we briefly mention here. These directives can be defined in terms of each other, and thus only a limited subset of hardware metrics must actually be defined in HDL; the remainder are formed by a simple macro syntax that can compose new metrics from other ones. The hardware subset includes directives such as conditional constructs (from basic to multicycle pattern-based conditions), sum, min/max, histogram, and several directives aiding in iteration which generate arrays of linear or exponential sequences; composite directives include items for correlation, additional multicycle pattern-based conditions, and even computing average and standard deviation. Software metrics could be similarly defined using C/C++ code modules.

Bottleneck rules include a description of the bottleneck; the bottleneck condition, which can be any boolean C/C++ expression, typically employing one or more metrics; textual suggestions for resolving the bottleneck along with arguments to insert into the text (given in printf-style format); whether the bottleneck applies to individual blocks or whether it can be applied to the entire application (or both); any additional metrics the user may be interested in; and information concerning the original time and new time if the bottleneck were resolved, which is used for computing speedup. Speedup is used to filter out bottlenecks that, if remedied, would not improve application performance by at least the user-defined speedup threshold. Bottleneck rules may be added or modified by the user through ReCAP's GUI and are saved along with all "reasons" in a separate file to facilitate sharing of bottleneck detection strategies in a community; the file format is currently that used by the standard Java "Properties" class, although other formats such as XML could also be of use.

ReCAP detects and produces reports concerning bottlenecks at runtime immediately after the user application has finished executing by testing all applicable bottleneck rules on all software and hardware blocks and for the application as a whole. We augmented ReCAP's SVG-based visualization from Koehler and George [2010] with warning icons for blocks containing bottlenecks. These icons directly link to an HTML file containing bottlenecks detected in that block; internal named anchors are used to jump within a single bottleneck file, and thus all bottlenecks may be reviewed

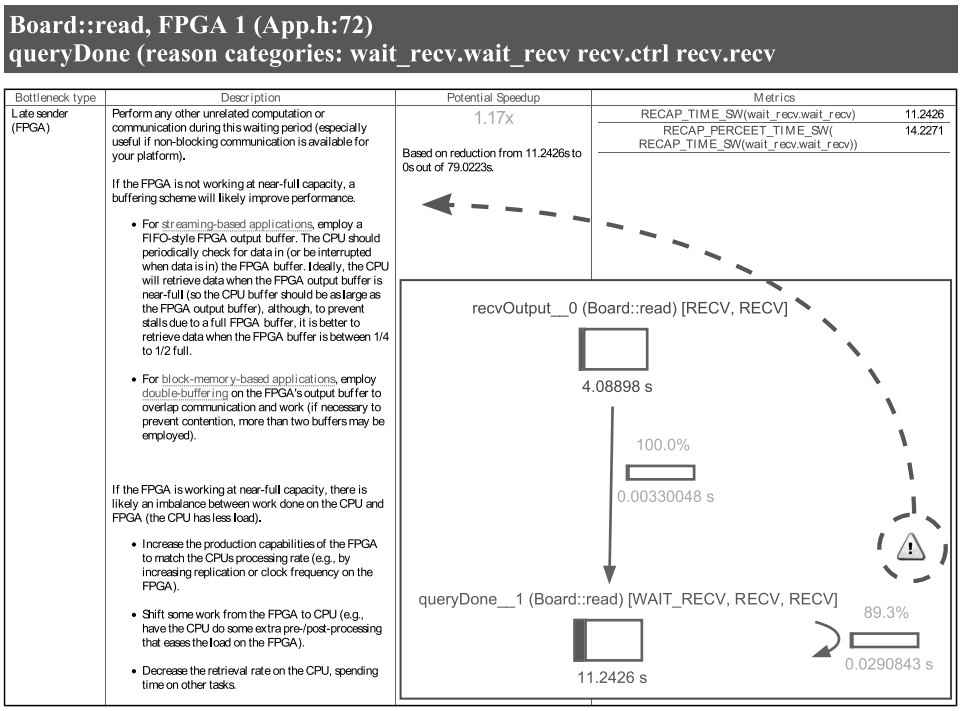


Fig. 6. Example of bottleneck detection results, showing inclusion of warning icons to indicate blocks with bottlenecks and a portion of the detailed bottleneck report.

directly as well. All reported bottlenecks display information concerning the bottleneck type; potential speedup if the bottleneck were remedied; suggestions for remedying the bottleneck, which can include specific data from tool metrics; all values of metrics used to determine whether to display this bottleneck; and any other user-specified metrics of interest. Currently, speedup presented is ideal, assuming no other bottlenecks prevent the application from improving by the given amount; in reality, dependencies amongst blocks could result in considerably less speedup. Our prior performance exploration work in Koehler and George [2010] dealt directly with this estimation problem; thus, while not discussed or implemented in this work, integrating performance exploration would significantly improve the accuracy of speedup estimates. Finally, platform-specific suggestions and microbenchmark data are also included, the latter of which is included as a link to an HTML table. Figure 6 shows an example of ReCAP's augmented visualization, which contains warning icons for blocks with bottlenecks detected, as well as the associated bottleneck information.

Although users are free to add or modify "reasons," "bottleneck rules," and platform templates, typical users of ReCAP need only add pragmas to their user source code to take full advantage of bottleneck detection; Section 5 will present our taxonomy of common RC bottlenecks and associated "reasons" that are provided by default in ReCAP. In addition, while we deal with only CPUs or FPGAs here, our block-based approach could be applicable to a broad class of heterogeneous systems (e.g., GPUs, DSPs, Cell processors); only the methodology for instrumentation and measurement must change.

As mentioned in the Introduction, our current implementation relies solely on profile data for bottleneck detection, even though traditional HPC employs trace data for this purpose; to compensate, ReCAP does provide a number of useful time-dependent profile

metrics, such as those reporting on consecutive API calls and transitions between different states in both software and hardware. While ReCAP supports tracing in both software and hardware, the largest FPGAs currently contain less than 10MB of on-chip memory; in contrast, a 40-block, 100MHz design generating 64-bit trace records every 10 cycles would require 3.2GB/s. Further, future devices with larger memories are likely to employ larger applications, which will likely generate larger amounts of trace data. Thus, reducing the number of trace events along with compressing or otherwise preprocessing trace data to save storage could be of particular use. Also, unlike with CPUs, it can be difficult to pause a user application in an FPGA in order to write trace data to memory; the FPGA may interact with external hardware in a timing-dependent manner, allowing a pause to miss critical data or become unsynchronized with another device. If handshaking were required between the FPGA and all external devices, pausing the user's application on the FPGA may be a viable method for recording trace data, albeit by incurring additional runtime overhead.

We finally note that ReCAP does currently employ a memory hierarchy to improve tracing capabilities; local trace buffers are constructed from on-chip memory and then fed into a single, per-chip trace collector using either internal or external memory, the latter of which is connected manually at this time. This approach allows high-bandwidth recording for small bursts of trace records for each block while providing storage for larger amounts of trace data. As tracing can provide a wealth of data useful for bottleneck detection, further research into efficient performance-based tracing methodologies for FPGAs could be of great use.

5. COMMON BOTTLENECKS IN RC APPLICATIONS

In this section we attempt to systematically explore and taxonomize potential RC bottlenecks, drawing upon our experience with RC applications as well as concepts and techniques from knowledge-based bottleneck detection in traditional HPC. Since an RC application may contain all the problems of a standard parallel application, traditional HPC bottleneck detection tools are quite beneficial in tuning the CPU portion of an RC application, allowing this work to be easily integrated with the significant amount of literature and tools already present for traditional HPC. In fact, since ReCAP builds upon PPW, it inherits all of PPW's software bottleneck analysis capabilities. Thus, we focus on bottlenecks that may occur due to CPU-FPGA communication and for bottlenecks within the FPGA, culminating in a taxonomy of potential RC application bottlenecks that includes detection and optimization strategies for each bottleneck and dovetails with traditional HPC knowledge-based bottleneck detection research. Figure 7 provides our taxonomy of possible bottlenecks, which will be discussed for the remainder of this section. Note that while this taxonomy contains common bottlenecks that apply to both software and hardware, these bottlenecks are defined separately in ReCAP to permit different detection and optimization strategies for the same bottleneck.

We break our discussion into four general bottleneck categories: communication (Section 5.1), synchronization (Section 5.2), internal overhead (Section 5.3), and imbalances (Section 5.4). Since a number of bottlenecks that follow are detected simply by determining if a significant portion of time for a block was spent performing tasks with an associated reason, where "significant" implies a possible speedup of more than a user-defined threshold if the problem were remedied, we will only highlight detection strategies that require additional conditions to be tested.

5.1. Communication Bottlenecks

Bottlenecks may occur during communication between blocks (recall that a block can be a VHDL process, Verilog always block, or software thread, and thus this includes communication between CPUs and FPGAs, blocks within an FPGA, and even between

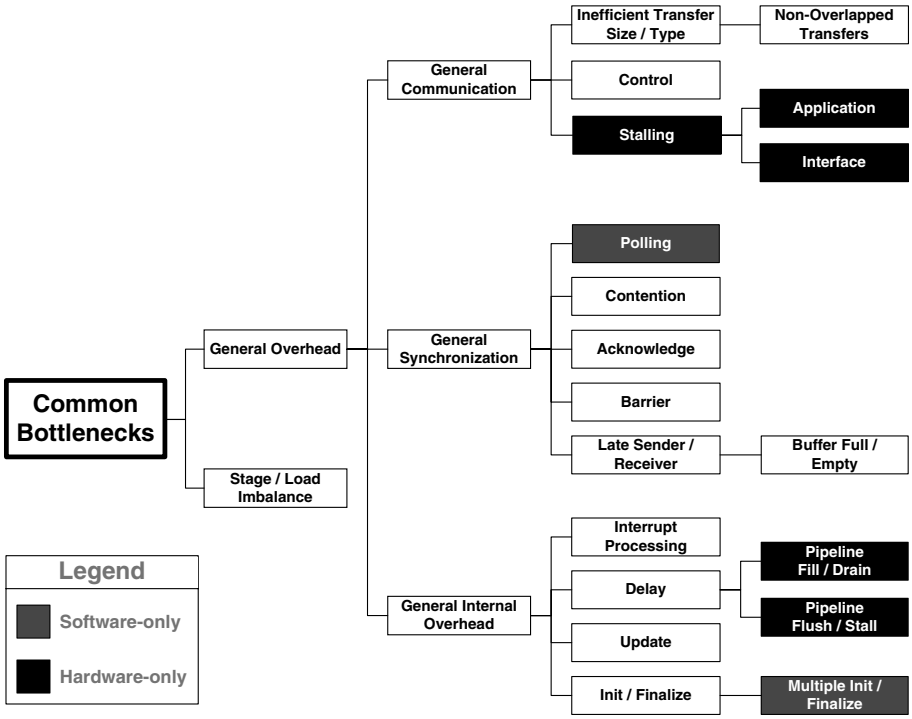
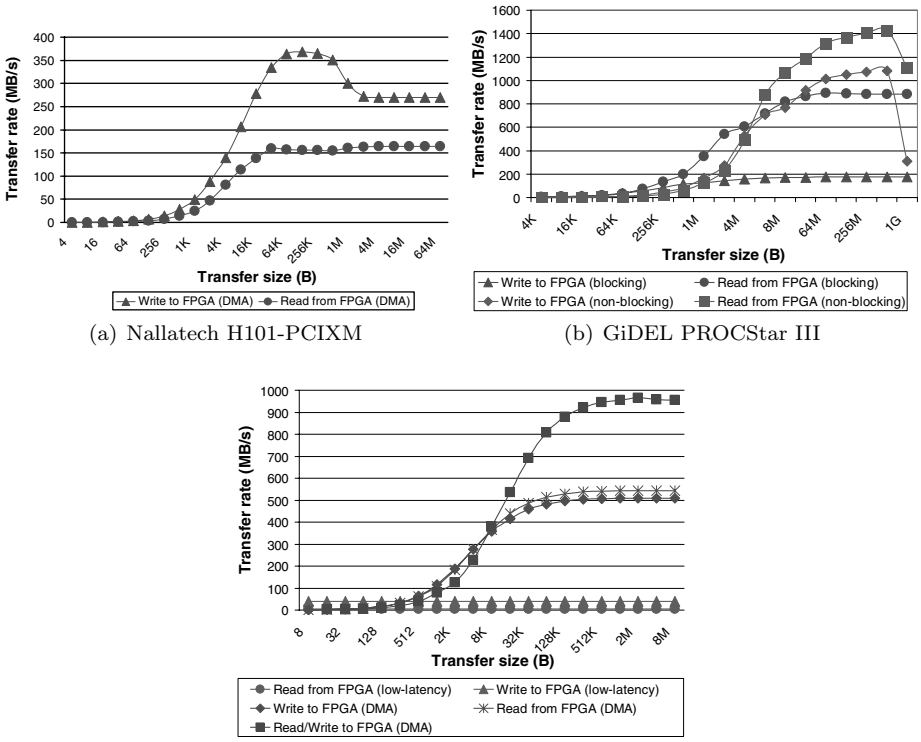


Fig. 7. Taxonomy of common bottlenecks in an RC system.

FPGAs). If a block spends a significant portion of time communicating with other blocks, this constitutes a potential *general communication* bottleneck. However, some blocks may be solely purposed for communication (especially common in FPGAs); in hardware, this case could be detected by searching for blocks with no pragmas specifying the "work" category, although, in software, ReCAP currently considers all time spent between API calls to be "work," making such detection more difficult. Suggestions for ameliorating this generic bottleneck include overlapping communication with other tasks if possible, employing bit-packing or compression, and repartitioning the algorithm to reduce the data transferred. Note that if more specific communication bottlenecks are detected in a block, basic bottleneck suggestions will still be displayed, thus eliminating the need to repeat these suggestions for more specific cases.

Another common scenario in communication is that transfer rates can vary drastically with transfer size and type depending on the protocols and interconnects used; see Figure 8 for a subset of transfer sizes and types for an XtremeData XD1000 platform, a Pentium-4 Xeon system equipped with a Nallatech H101-X accelerator, and a quad-core Xeon E5520 system equipped with two quad-FPGA GiDEL PROCStar III cards. Thus, using an *inefficient transfer type or size* constitutes a potential bottleneck. ReCAP currently only detects this bottleneck in software, since hardware communication is often not packetized and thus incurs no overhead beyond the actual data transfer, although more complicated protocols could be used in hardware for inter-FPGA communication if the FPGAs were not tightly coupled. For example, Figures 8(a) and 8(b) demonstrate the potential for large differences between read and write performance, whereas Figure 8(c) shows how different transfer types may perform better for different transfer sizes. Figure 8(b) also shows that *nonoverlapped transfers*, where multiple simultaneous transfers can occur via either nonblocking communication or collective



(a) Nallatech H101-PCIXM (b) GiDEL PROCStar III

(c) XtremeData XD1000

Fig. 8. Platform transfer rate versus transfer size across various RC systems and communication types, demonstrating common communication problems. The nonblocking transfer rate was computed as the sustained cumulative transfer rate of eight concurrent nonblocking transfers.

functions (e.g., broadcast, scatter, gather, reduce), can be a potential bottleneck in RC applications; this is also true in hardware where it is ideal for a block to perform as much communication as possible in parallel with other tasks, and thus communication should be overlapped wherever possible. These phenomena have been well-researched in non-RC contexts [Alexandrov et al. 1995].

ReCAP detects such bottlenecks by comparing actual bandwidth recorded with the best microbenchmark bandwidths from the platform template; as mentioned in Section 3, ReCAP could also automatically perform microbenchmarks or detect performance deviations between similar API calls. From this data, ReCAP can make specific suggestions on the best type to use if transfer size was held constant, the best size to use if transfer type was held constant, and the best overall size and type to use for the platform. The best size is given as a range of transfer sizes that achieve, for example, 95% of maximum bandwidth for that transfer type along with a link to microbenchmark data provided for further investigation. This type of specific suggestion, along with potential speedups, is just one example of where ReCAP's suggestions can be useful, even for expert RC designers. Suggestions for increasing transfer size include unifying different transfers together, such as by placing different items consecutively in the memory map or by embedding control, masking, or address information in the communication stream; note that an increase in transfer size may necessitate creating or increasing the size of buffers to handle larger transfers if the data cannot be processed in real

time[4]. Transfer size may be decreased simply by breaking a large transfer into pieces, in which case buffers may be able to be reduced in size to save resources. As an example, transfering thirty-two 128KB packets on the Nallatech platform is more efficient than transfering a single 4MB packet, yielding 360MB/s and 291MB/s, respectively.

Control messages deserve specific mention as these are commonly employed in RC applications and typically incur significant overhead. ReCAP detects control messages directly from pragmas that specify communication to be control oriented. One suggestion for addressing control bottlenecks includes moving as much control logic onto the FPGA as possible. For example, instead of reading from an FPGA register, testing a value, and then possibly starting some action on the FPGA, the register test can be moved onto the FPGA. Additional suggestions include employing unused address bits in a read to carry control information or increasing the size/duration of the work being controlled (e.g., loop unrolling), thus reducing the amount of control needed.

In software and hardware, it may be possible for both the application and interface to stall a transfer, such as when no data is available or when another transfer preempts the first transfer, potentially causing a *stalling* bottleneck on behalf of either the *application* or *interface*. These bottlenecks can be detected in hardware by monitoring the appropriate signals on the interface port; however, it is rare that an API would surface such information in software, and stalling could occur in some intermediate buffer not easily monitored from either software or hardware. These bottlenecks could be remedied by the addition of buffers to prevent stalling or, in the case of the application, by increasing the rate at which the application can accept data, such as via replication of application components. While a similar suggestion could be made for improving an API, this is typically not possible for an application designer, who often does not have the source code for an API available, and thus reducing the application's data processing speed is suggested as this is unlikely to affect performance and may save resources or power.

5.2. Synchronization Bottlenecks

Synchronization bottlenecks indicate a block is waiting on one or more other blocks, either to transfer data or to reach a certain point in execution before continuing. If two blocks must communicate synchronously, one block may arrive to its side of communication later than another, permitting a *late sender* or *late receiver* bottleneck (common terminology in traditional HPC bottleneck detection). Suggestions for remedying this situation include performing unrelated computation or communication while waiting, buffering or double-buffering depending on whether the application is streaming-based or block-memory-based, or remedying an imbalance between blocks either by increasing the efficiency of the slower block or decreasing the efficiency of faster blocks, the latter of which could reduce power or resources used. If employing a buffer in the streaming case, ReCAP suggests a transmission size that is at most half the total buffer size to ensure the buffer can be refilled before it drains and at least a quarter of the total buffer size to maximize communication bandwidth.

As buffering is a common technique for addressing synchronization bottlenecks, additional common bottlenecks include a *full buffer* or *empty buffer* bottleneck. ReCAP suggests an imbalance may exist, as mentioned before, or that burst traffic may be occurring if the buffer is full and empty often, in which case the buffer size should be increased to handle larger bursts or the transmission size should be decreased and

[4]ReCAP's actual suggestions often contain additional description and examples not included here for brevity, but which are quite useful for less-experienced RC designers.

the transmission frequency increased to smooth out the bursts, such as by having a separate thread handle transmission periodically.

Several additional types of synchronization bottlenecks are also handled in ReCAP. A *polling* bottleneck can occur when status of another block is repeatedly queried until some condition is met. In hardware, this situation is extremely common and is similar to a normal synchronization bottleneck; however, in software, this behavior is problematic, wasting communication bandwidth and processor time. ReCAP detects this bottleneck by searching for any API call that is waiting to receive data and is, on average, executed three or more times consecutively. Suggestions include using interrupts if supported by the system and API, moving any unrelated tasks between polling calls, and using a separate thread to perform the poll, potentially adding an indicator into data returned from the poll, such as "percent done," to estimate time the thread should sleep before polling again.

An *acknowledge* bottleneck can occur when waiting for a block to acknowledge some event. In this case, ReCAP suggests specutively continuing without the acknowledge if an acknowledge is expected, storing any checkpoint or state information needed to restart or retry a task if the acknowledge does not arrive so long as this storage is not prohibitive in terms of memory requirements. A *barrier* bottleneck indicates that a block is waiting for other blocks to reach a given point of execution before continuing. In this case, decoupling these blocks via buffering may be possible if no feedback or resource contention exists that would prevent blocks from continuing. For example, to average the results from several blocks, a running average could be computed or a buffer could be added to the output of each block, with the average being computed from the buffer output.

A *contention* bottleneck detects when blocks are spending a significant time waiting on a shared resource. There are a number of suggestions for alleviating this type of bottleneck. One suggestion involves reducing time needed to acquire or release a lock, usually by adding an arbiter that manages all requests to the shared device. Another suggestion includes increasing the efficiency or replication of the shared resource itself. For example, placing a memory in a faster clock domain, such as one that can handle two transactions per application cycle, can reduce contention. Similarly, replication of the shared resource may reduce contention as well depending on the coupling required amongst replicated copies. For example, replicating a memory in hardware or using an additional memory bank from software permits multiple independent reads to be issued simultaneously to different copies of memory at the expense of ensuring a write is issued to all memories for consistency. Additional suggestions include forcing a staggered ordering to reduce or eliminate locking, increasing (or decreasing) granularity of tasks performed between lock and release if the cost for locking is high (or low), ensuring no block holds a shared resource any longer than necessary, ensuring the minimum locking is performed to still ensure consistency, and finally reducing the efficiency of other blocks or the number of blocks accessing the shared resource to potentially save resources with little performance loss.

5.3. Internal Overhead Bottlenecks

A block can spend a significant amount of time performing bookkeeping or other internal overhead tasks, thus allowing an *internal overhead* bottleneck to occur. As with other generic bottlenecks, this bottleneck may be addressed by parallelizing, pipelining, or otherwise overlapping these tasks with others or by increasing the size/duration of the work associated with internal overhead so that fewer overhead tasks are performed (e.g., loop unrolling). However, many specific variants of this bottleneck are also detected. *Initialization*, *finalization*, and *update* bottlenecks occur when significant time is spent in a block performing initialization, update, or finalization tasks.

Suggestions include reducing or eliminating this overhead; for example, if a histogram is to be accumulated in memory and thus needs to be cleared for each new dataset, it may be possible to instead maintain a bit-vector that indicates which memory locations have been accessed since the last dataset and thus determine whether to store or add a given value to the current memory location. A *multiple initialization* or *multiple finalization* bottleneck indicates that software has not only spent a significant amount of time configuring or releasing the FPGA, but that software performed this task several times; this typically indicates several different configuration files have been loaded during runtime to accelerate different phases of an application. These bottlenecks are detected by ensuring the API call is of the appropriate type (e.g., configure, release) and called at least twice. Suggestions for optimization include adding functionality to, or generalizing functionality in, each configuration file to reduce the number of configurations, rescheduling the CPU's work if the same configuration file is loaded multiple times so that overhead from reprogramming the same file is reduced, and repartitioning the algorithm to minimize the amount of functionality needed by the FPGA, such as by moving some pre- or postprocessing tasks from the FPGA to the CPU.

An *interrupt-processing* bottleneck may occur if a block is interrupted too often by another block. Note that while the physical interrupt is communication, this refers to interrupt handling, and thus is an internal task. In this case, the number of exceptional circumstances that cause interrupts should be reduced, such as by increasing precision to reduce overflow interrupts or by resolving the most common interrupts locally, if possible. A *delay* bottleneck occurs when a block must delay for some internal reason, such as when waiting for an extra-long combinatorial path or internal pipeline, the latter of which is handled specifically in the *pipeline fill/drain* and *pipeline flush/stall* bottlenecks. Delay bottleneck suggestions focus on reducing the latency causing the delay or overlapping these latencies with other delays or useful work. Suggestions for addressing a pipeline fill/drain bottleneck include filling a pipeline with the next dataset while still processing or draining the previous dataset. Suggestions for addressing a pipeline flush/stall bottleneck include decreasing latency or pipeline stages, although this must be balanced with the effect on the FPGA's maximum frequency; moving detection of flush conditions earlier to minimize stages flushed; and having pipelines with a large number of stalls process data from multiple streams, interleaving independent data into the pipeline to minimize dependency stalls.

5.4. Imbalance Bottlenecks

An *imbalance* of computation between two or more related blocks is also considered a potential bottleneck. A *stage imbalance* refers to an imbalance where a block depends on other blocks, such as in a pipeline. Optimization of this bottleneck involves improving the performance of slower blocks, such as through additional stage division or replication, or by reducing the performance of faster blocks to potentially save resources. A *load imbalance* indicates an imbalance between parallel blocks that receive data, potentially indirectly, from the same source block, such as is common with replicated cores. One possible optimization for this bottleneck includes improving the data distribution scheme. For example, employing a look-ahead round-robin scheme that determines whether any of the next four or eight blocks are idle, skipping them all if so, or employing a priority-based selection scheme such as least-recently used can often improve performance. Additional optimizations include adding input buffers or changing the replication factor to possibly distribute data more evenly. For example, simply using a prime replication factor may distribute data more evenly than a heavily composite replication factor.

ReCAP detects both bottlenecks by searching through the block dependency graph given by user pragmas to determine where potential bottlenecks are located; a number

Table I. RC Platforms Employed During Case Studies

Name	CPU(s)	FPGA(s)	API type
Nallatech H101-PCIXM	Pentium 4 Xeon 3.2GHz	One Virtex 4 LX100 (PCI-X card)	C, simple memory
XtremeData XD1000	Dual-core Opteron 285 2.6GHz	One Stratix II S180 (HyperTransport socket)	C++, DMA
GiDEL PROCStar III	Quad-core Xeon 5520 2.26GHz	Four Stratix III E260s (PCIe card)	C++, simple memory

of metrics are provided that return statistics over these block groups to facilitate such bottleneck detection. Within the provided suggestions, specific data about the best and worst performing blocks as well as the average performance of all related blocks are also given to allow a designer to determine how severe the imbalance is and what techniques may be best in addressing the bottleneck.

6. CASE STUDIES

To demonstrate the utility of RC bottleneck detection and platform templates, our extended ReCAP tool was employed on two different applications on a total of three diverse RC platforms: a time-domain finite impulse response benchmark [Haney et al. 2005] on a GiDEL PROCStar III [GiDEL 2010] and a two-dimensional probability density function estimator application [Nagarajan et al. 2008] on both a Nallatech H101-PCIXM card [Nallatech 2010] and an XtremeData XD1000 system [XtremeData Inc. 2010]. Table I provides details for these RC systems.

6.1. Time-Domain Finite Impulse Response

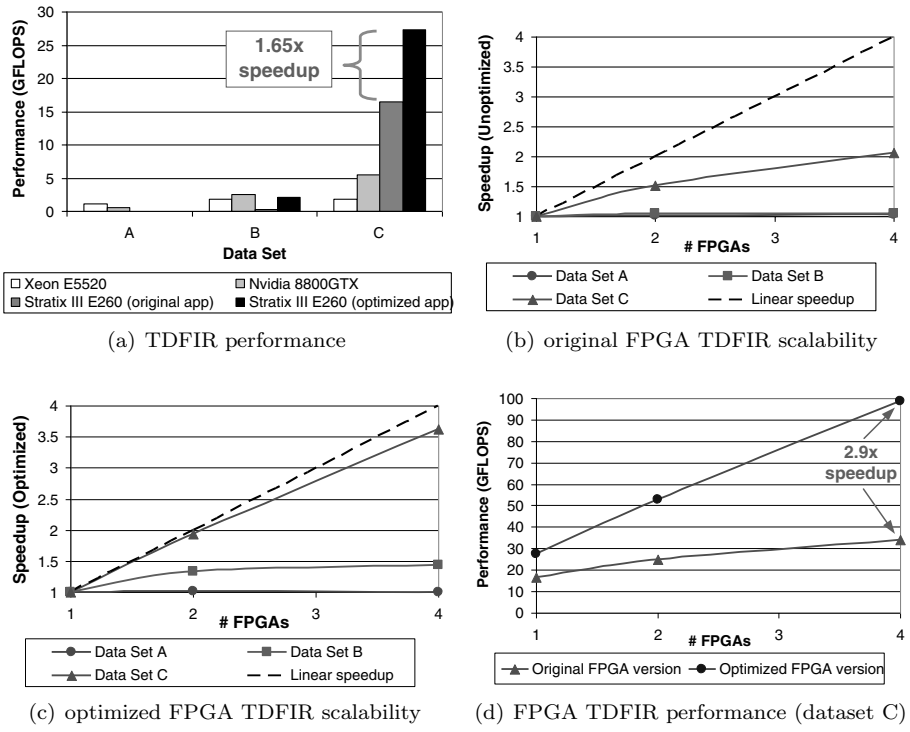
The Time-Domain Finite Impulse Response (TDFIR) benchmark is part of the HPEC challenge benchmark suite [Haney et al. 2005] and has been accelerated on GPUs as well [McGraw-Herdeg et al. 2007]. For an FPGA-accelerated version, we implemented convolution for real numbers rather than complex; however, for consistency of results, we report all numbers in GFLOPS, thus accounting for the fact that each basic computation involves only two floating-point operations rather than eight for the complex versions. This case study was performed on the GiDEL system (Table I) using Quartus 9.1SP2 and GCC 4.4.3 with −O3 optimization; all times given are the average of three executions. The TDFIR benchmark was able to execute at 125MHz on the FPGA for both the original and optimized versions, and thus all executions were performed at this frequency for uniformity of results. All FPGA benchmark execution times include all data transfer times between the CPU and FPGA as well as any other needed CPU tasks such as data movement; we only exclude the FPGA initialization/finalization time, since the configuration file could be preloaded once and then used indefinitely, such as for streaming large amounts of data through.

Three datasets were used for evaluation, which consisted of random data with the same kernel size, input size, and iterations (i.e., the number of different subdatasets with the given kernel and input size that must be computed). Dataset A and B are the standard datasets given in the HPEC challenge, while dataset C is the largest dataset from McGraw-Herdeg et al. [2007] (Table II). We compare FPGA results from a Stratix III E260 to both the Intel Xeon E5520 processor (the host processor in the GiDEL system) and the results given in McGraw-Herdeg et al. [2007] for an NVIDIA 8800GTX.

Upon executing the FPGA version of TDFIR, two problems were observed. First, the single-FPGA version was slower than either the CPU or GPU version for the first two datasets, although the FPGA version did achieve a 9.1x speedup over the CPU and a 3.0x speedup over the GPU on dataset C (Figure 9(a)). Second, when scaling up from

Table II. Datasets Evaluated for TDFIR Benchmark

Dataset	Kernel Size	Input Size	Iterations
A	12	1024	20
B	128	4096	64
C	4096	32768	128



(a) TDFIR performance

(b) original FPGA TDFIR scalability

(c) optimized FPGA TDFIR scalability

(d) FPGA TDFIR performance (dataset C)

Fig. 9. TDFIR performance on various devices (including both the original and optimized FPGA performance after bottleneck detection.

1 to 4 FPGAs, performance scaled poorly, resulting in less than 1.06x speedup for datasets A and B and less than 2.1x speedup for dataset C, even though no communication or synchronization was needed between FPGAs; different iterations were simply scattered to the different FPGAs (Figure 9(b)).

We then employed ReCAP's automatic bottleneck detection on the FPGA version of TDFIR, using dataset C for execution. Overhead for obtaining performance data was acceptable, incurring an additional 5.1% in software time, 1.8% of FPGA logic resources, 0.8% of FPGA register resources, and a negligible (less than 1%) decrease in frequency; the benchmark still met the required 125MHz. ReCAP detected several communication bottlenecks in software including "inefficient communication size and type" with an ideal speedup of 11.63x; "control overhead" with an ideal speedup of 3.60x; a "late sender" bottleneck for the FPGA, where the CPU is ready to receive data but the FPGA is late in sending that data, with an ideal speedup of 2.52x; and a "late sender" bottleneck for the CPU, where the FPGA is ready to receive data but the CPU is late in sending that data, with an ideal speedup of 1.63x. As mentioned earlier, the speedup numbers are ideal, and thus our performance exploration framework in Koehler and George [2010] should be used for more accurate predictions. For example, while ReCAP suggests earlier that an 11.63x speedup is possible if the associated

bottleneck is remedied, ReCAP's visualization shows the FPGA is working 54.1% of the time when processing dataset C, thus limiting speedup to at most $1/0.541 = 1.85x$; the FPGA is working much less of the time for datasets A and B and thus better speedup is possible for these cases. Nonetheless, the ideal speedup numbers given are still useful for providing an upper bound on potential performance improvement as well as for serving as a severity indicator for a given bottleneck, thus indicating which bottlenecks should typically be addressed first.

We first chose to address the "inefficient communication size and type" bottleneck. For one API call, bottleneck detection indicated a 2.52x speedup was possible if the transfer size were increased to between 32MB and 64MB (or higher for asynchronous communication) while a 2.39x speedup was possible by switching to low-latency transfers. As changing the transfer size would be more difficult and yet not result in much additional performance beyond that gained by switching the transfer type, we chose the latter approach for this API call. However, for several other API calls, solely switching to a different communication type was not recommended by ReCAP, and thus we increased the transfer size by a factor of 8 to 10 via buffers and logic to handle batch transfers[5]. These changes alone resulted in a 1.47x improvement in performance for dataset C on one FPGA and a 1.68x performance improvement for 4 FPGAs. However, with only a 2.4x speedup between the 1- and 4- FPGA versions, scalability was still low.

We next addressed the "late sender" bottlenecks, which suggested overlapping the waiting period with other computation or communication, using asynchronous transfers if available, as well as employing double or higher-order buffering. Thus, we overlapped communication using asynchronous transfers and quadruple buffering with three different external memories as well as internal memory, providing an additional 1.12x improvement in performance for dataset C on one FPGA and an additional 1.73x performance improvement for 4 FPGAs; the 1-FPGA version experienced less performance increase due to heavy use of its computational units. As shown in Figure 9(c), the second optimization resulted in a scalability of over 90% of the ideal (3.61/4.00) for dataset C and noticeable improvement in scalability for dataset B.

Thus, by employing these two optimizations, performance was improved by 1.65x for the single-FPGA version and 2.9x for the 4-FPGA version, achieving 27.3 GFLOPS and 98.9 GFLOPS on dataset C, respectively. The 4-FPGA version achieved a total of 54.4x speedup over the software baseline. Final performance for each dataset is shown as black bars in Figure 9(a) and with circle markers in Figure 9(d). Further, the single FPGA version now performed 1.15x better than the CPU on dataset B, whereas the original version had performed 6.7 times slower than the CPU. While performance on dataset A was increased by 5.7x, the FPGA continued to perform poorly due to communication overhead; the Nvidia 8800GTX experienced a similar, albeit less pronounced, effect for this dataset, as seen in Figure 9(a). Further improvements suggested by ReCAP, such as moving a memory clear operation from software into FPGA logic, would likely achieve some additional speedup, although the FPGA cores were observed to be working 68.7% of the time in the 4-FPGA version, limiting the amount of further speedup possible without employing additional resources or a better algorithm. It is noteworthy that given the reported bottlenecks and optimization suggestions, actual optimizations were made within two days, resulting in higher performance as

[5]ReCAP suggested increasing the transfer size much further for optimal transmission, which was infeasible due to memory limitations. Thus, we manually determined a balance between significantly better bandwidth from the microbenchmark table, memory overhead, and achieving even divisibility into the number of iterations for each dataset; the last requirement resulted in a batch size of 10 for dataset A since 8 does not divide evenly into the stipulated 20 iterations for that dataset.

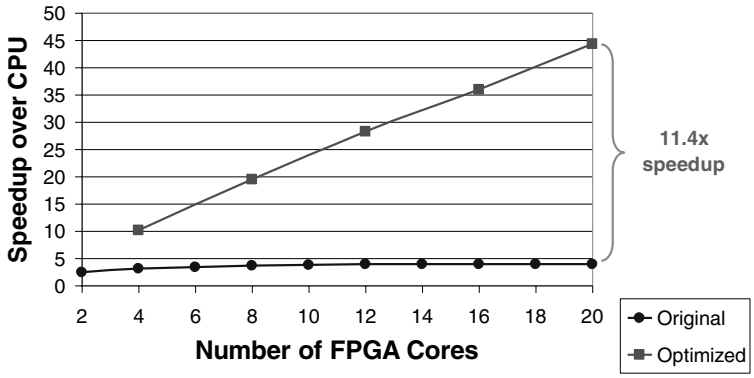


Fig. 10. Speedup of initial and improved versions of the 2DPDF application when compared to a CPU baseline.

well as improved productivity compared to manual, ad hoc bottleneck location and optimization.

6.2. Two-Dimensional Probability Density Function Estimation

The two-dimensional Probability Density Function (2DPDF) estimation application is used in various engineering, financial, and scientific fields where nonparametric probabilistic approaches are required; the application is computationally intensive, involving $O(m \times n^2)$ operations where m is the number of sample points and n is the number of bins per dimension. Our implementation uses the Parzen-window algorithm and a fixed-point format ([18,9] external precision, [48,18] internal precision, given in [total, fractional] format) [Nagarajan et al. 2008].

Our experimental setup consisted of the Nallatech system (Table I), using GCC 4.4.3 and Xilinx ISE 11.5 to compile all C and VHDL files, respectively. A software baseline was written in C using only integer arithmetic to better compare to the FPGA's fixed-point format, as the CPU's floating-point version was slower. This baseline was compiled with −O3 optimization and executed on the attached Pentium 4 Xeon 3.2GHz processor; all execution times were computed from the average of 3 executions. The 2DPDF application was capable of operation at 100MHz or higher for all design variants, and thus 100MHz was used for uniformity of performance results. Similar to the convolution case study, all FPGA application execution times include all data transfer times between the CPU and FPGA as well as any other needed CPU tasks such as data movement; we only exclude the FPGA initialization/finalization time, since the configuration file could be preloaded once and then used indefinitely, such as for streaming large amounts of data through.

Initially, we attempted to gain speedup by extending the 2DPDF application to a multicore design within the FPGA. Figure 10 shows the initial execution times for several multicore variants (circle markers). While our software baseline required 250.5 seconds to process 1,024,000 points, the 20-core FPGA design required only 64.5 seconds for the same dataset, resulting in a 3.9x speedup. However, Figure 10 demonstrates that these additional cores provided diminishing performance improvements.

We then employed ReCAP's automatic bottleneck detection on the 20-core design. Overhead for obtaining performance data was acceptable, incurring an additional 14.2% in software time due to millions of API calls during execution, 4.3% of FPGA logic resources, 2.3% of FPGA register resources, and a maximum 2.2% decrease in frequency. ReCAP identified a number of potential bottlenecks including a "CPU late sender" bottleneck with ideal 16.21x speedup, a "control" overhead bottleneck with

ideal 1.55x speedup, an "inefficient communication size/type" bottleneck with ideal 9.83x speedup, a "control" overhead bottleneck with ideal 1.38x speedup, and an "FPGA late sender" bottleneck with ideal 1.32x speedup. Specifically, bottlenecks were also detected in five of the twelve individual API calls in software involving clearing memory, starting and stopping cores, checking to see if cores were complete, and reading the output; each API call's potential speedup ranged from 1.15x to 1.34x. The generic "all overhead" bottleneck on the FPGA indicated that, if the FPGA were fully utilized, a possible 16.81x speedup could be achieved.

Based on the optimizations suggested earlier, we focused on consolidating the large number of small transfers performed and on moving control logic onto the FPGA. Specifically, several input buffers were increased in size from 2KB to 32KB, the minimum ideal transfer size suggested by ReCAP; output buffers were placed consecutively in the memory map, permitting a single larger read rather than several smaller reads; register data was consolidated to reduce the amount of data polled; and control logic was moved onto the FPGA, performing tasks such as automatically clearing intermediate FPGA buffers when receiving new data rather than relying on software to manually control this process.

Speedup for the improved 2DPDF application increased from 3.9x to 44.4x when compared with the software baseline, resulting in an 11.4x performance improvement between the unoptimized and optimized versions (square markers in Figure 10) and demonstrating far more linear speedup with respect to the number of cores employed. Interestingly, we also discovered that the new version incurred slightly less rounding error since FPGA data is rounded when transferred to the CPU and fewer transfers were employed in the optimized version, although the error incurred by the original version was deemed acceptable. Again, it is noteworthy that given the reported bottlenecks and optimization suggestions, actual optimizations were made within a day, resulting in a significant performance increase as well as improved productivity compared to manual approaches for bottleneck detection and optimization.

As a final attempt to improve performance, we ported the optimized 2DPDF application to the XD1000 platform (Table I), which allowed us to increase computational resources by a factor of 2.4x. Software and hardware were compiled with GCC 4.3.2 with -O3 optimization and Quartus 9.1SP2, respectively. We obtained an additional 2.5x speedup over the Nallatech implementation; note that a speedup greater than 2.4x is possible due to faster transfers afforded by the HyperTransport interconnect on the XD1000. We used ReCAP's automatic bottleneck detection on the ported application, incurring an additional 5.2% of software runtime overhead, 4.7% of FPGA logic resources, 1.8% of FPGA register resources, and a 15.3% frequency degradation, which resulted from the application filling 87% of the device before instrumentation, and thus 92% of the device after instrumentation; the 100MHz requirement was still met by the instrumented version.

ReCAP's bottleneck detection returned several potential bottlenecks, but the ideal speedups were much lower than in the previous cases. For example, in software, all speedups were less than 4.2x; in hardware the cores showed speedup potential of at most 1.31x, with a stage imbalance bottleneck as the exception. While this stage imbalance bottleneck showed a potential of up to 5.5x speedup, the stage imbalance it referred to involved a basic distribution core that performs very little communication and no work; with the integration of the aforementioned performance exploration framework, such false positives could be significantly reduced. Thus, given lackluster performance improvements predicted, we chose not to optimize further, although there are scenarios where a user may believe the potential speedup warrants additional effort, such as if the application must be executed many times or if runtime were significantly longer. This result underscores an often overlooked benefit of bottleneck detection; bottleneck

Table III. 2DPDF Results for Both the Nallatech and XD1000 Platforms

Application Version	Runtime (s)	Speedup
Pentium 4 Xeon CPU	250.515	1.0
Nallatech (FPGA original)	64.506	3.9
Nallatech (FPGA optimized)	5.648	44.4
XD1000 (FPGA)	2.218	112.9

Speedup is given with respect to the software baseline executed on a Pentium 4 Xeon 3.2GHz CPU.

detection can often be as useful in indicating what not to optimize as it is in what to optimize. Table III gives execution times for the 2DPDF application on both platforms, showing that the XD1000 version achieved a 112.9x speedup over the original Pentium 4 Xeon 3.2GHz software baseline.

7. CONCLUSIONS

In this article, we proposed what we believe to be the first automatic bottleneck detection framework and tool for RC applications, including a framework for platform templates that permits more accurate, platform-aware bottleneck detection as well as tool portability across diverse RC systems. These templates are easily created by end-users, typically in a few hours provided the platform fits within the generic platform-template model. In addition, we formulated what we believe to be the first taxonomy of common bottlenecks for RC applications, along with associated detection and optimization strategies for each of these bottlenecks, to populate ReCAP's knowledge base for bottleneck detection. Our bottleneck knowledge base is extensible, providing for user detection of bottlenecks not envisioned by the authors. The knowledge-based bottleneck detection framework and platform-template system were implemented by extending our Reconfigurable Computing Application Performance (ReCAP) tool, providing users with a full-featured RC performance analysis tool for diverse RC systems that can significantly accelerate the optimization process.

We then demonstrated bottleneck detection in ReCAP via two case studies involving a Time-Domain Finite Impulse Response (TDFIR) benchmark from the HPEC challenge and a two-dimensional Probability Density Function (2DPDF) estimation application on a total of three diverse platforms. ReCAP reported a number of bottleneck types discovered in both software and hardware along with optimization suggestions and potential speedup for each bottleneck type. Several of these optimization suggestions were employed to achieve an additional 2.9x speedup for TDFIR, resulting in a total 54.4x speedup over the CPU baseline, and an additional 11.4x speedup for 2DPDF estimation, resulting in a 44.4x speedup over the CPU baseline. We ported 2DPDF to the XD1000 platform, which provided an additional 2.5x speedup due to increased computational resources. Based on lackluster potential speedup reported by bottleneck detection, we did not further optimize the application, resulting in a total 112.9x speedup over the Pentium 4 Xeon 3.2GHz CPU baseline.

Future work includes the implementation and integration of our performance exploration framework (detailed in Koehler and George [2010]) to provide better predictions for expected speedup if a bottleneck were addressed; the expansion of our metric framework to allow users the ability to define their own metrics; research into the inclusion of trace-based analyses for RC bottleneck detection; the extension of our platform-template system to handle memory and other external resources to the FPGA besides the currently supported communication port to the CPU; and completion of support for Verilog-based applications. Finally, similar research is also needed in bottleneck detection for RC applications that use HLLs to describe hardware (commonly referred

to as High-Level Synthesis, or HLS), as these languages are increasing in importance and popularity.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Dr. Karthik Nagarajan for the initial implementation of the 2D-PDF estimator application.

REFERENCES

AGGARWAL, V., GARCIA, R., STITT, G., GEORGE, A., AND LAM, H. 2009. SCF: A device- and language-independent task coordination framework for reconfigurable, heterogeneous systems. In *Proceedings of the 3rd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA'09)*. ACM, New York, 19–28.

ALEXANDROV, A., IONESCU, M. F., SCHAUSER, K. E., AND SCHEIMAN, C. 1995. LogGP: Incorporating long messages into the logp model—One step closer towards a realistic model for parallel computation. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*. ACM, New York, 95–105.

BARROSO, L. A. 2005. The price of performance. *Queue 3,* 7, 48–53.

BODENNER, R. 2010. Creating platform support packages. http://www.impulseaccelerated.com/AppNotes/APP109_PSP/IATAPP109_PSP.pdf.

CHAMBERLAIN, R., FRANKLIN, M., TYSON, E., BUCKLEY, J., BUHLER, J., GALLOWAY, G., GAYEN, S., HALL, M., SHANDS, E., AND SINGLA, N. 2010. Auto-Pipe: Streaming applications on architecturally diverse systems. *Comput. 43,* 3, 42–49.

CHE, S., LI, J., SHEAFFER, J. W., SKADRON, K., AND LACH, J. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the Symposium on Application Specific Processors (SASP'08)*. IEEE Computer Society, Los Alamitos, CA, 101–107.

CHUNG, I.-H., CONG, G., KLEPACKI, D., SBARAGLIA, S., SEELAM, S., AND WEN, H.-F. 2008. A framework for automated performance bottleneck detection. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. 1 –7.

CRAWFORD, C. H., HENNING, P., KISTLER, M., AND WRIGHT, C. 2008. Accelerating computing with the cell broadband engine processor. In *Proceedings of the Conference on Computing Frontiers*. ACM, New York, 3–12.

CRAY. 2010. Cray XD1 datasheet. http://www.hpc.unm.edu/%7Etlthomas/buildout/Cray_XD1_Datasheet.pdf.

CURRERI, J., KOEHLER, S., GEORGE, A. D., HOLLAND, B., AND GARCIA, R. 2010. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Trans. Reconfig. Technol. Syst. 3,* 1, 1–23.

DEHON, A., ADAMS, J., DELORIMIER, M., KAPRE, N., MATSUDA, Y., NAEIMI, H., VANIER, M., AND WRIGHTON, M. 2004. Design patterns for reconfigurable computing. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 13–23.

GARCIA, P., COMPTON, K., SCHULTE, M., BLEM, E., AND FU, W. 2006. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embed. Syst. 1*, 13–13.

GIDEL. 2010. GiDEL PROCStar III PCIe x8™ computation accelerator. http://www.gidel.com/pdf/PROCStarIII%20Product%20Brief.pdf.

HANEY, R., MEUSE, T., KEPNER, J., AND LEBAK, J. 2005. The HPEC challenge benchmark suite. In *Proceedings of the 9th Annual High-Performance Embedded Computing Workshop (HPEC'05)*.

JORBA, J., MARGALEF, T., AND LUQUE, E. 2008. *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer (Chapter Search of Performance Inefficiencies in Message Passing Applications with KappaPI 2 Tool), 409–419.

KOEHLER, S., CURRERI, J., AND GEORGE, A. D. 2008. Performance analysis challenges and framework for high-performance reconfigurable computing. *Parall. Comput. 34,* 4-5, 217–230.

KOEHLER, S. AND GEORGE, A. D. 2010. Performance visualization and exploration for reconfigurable computing applications. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*.

LAUDON, J. 2005. Performance/watt: the new server focus. *SIGARCH Comput. Archit. News 33,* 4, 5–13.

MCGRAW-HERDEG, M. P., ENRIGHT, D. P., AND MICHEL, B. S. 2007. Benchmarking the NVIDIA 8800GTX with the CUDA development platform. In *Proceedings of the 11th Annual High-Performance Embedded Computing Workshop (HPEC'07)*.

MOHR, B. AND WOLF, F. 2003. *Euro-Par 2003 Parallel Processing*. Springer (Chapter KOJAK A Tool Set for Automatic Performance Analysis of Parallel Programs.) 1301–1304.

NAGARAJAN, K., HOLLAND, B., SLATTON, C., AND GEORGE, A. D. 2008. Scalable and portable architecture for probability density function estimation on FPGAs. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM'08)*. IEEE Computer Society, Los Alamitos, CA, 302–303.

NALLATECH. 2010. H101-PCIXM PCI-X FPGA accelerator card. http://www.nallatech.com/PCI-Express-Cards/h101-pcixm.html.

OPENFPGA. 2010. OpenFPGA GenAPI version 0.4 draft for comment. http://www.openfpga.org/Standards%20Documents/OpenFPGA-GenAPIv0.4.pdf.

SU, H.-H., BILLINGSLEY III, M., AND GEORGE, A. D. 2011. Parallel performance wizard: A performance system for the analysis of partitioned global-address-space applications. *Int. J. High-Perform. Comput. Appl.* in press.

SU, H.-H., BILLINGSLEY III, M., AND GEORGE, A. D. 2009. A distributed, programming model-independent automatic analysis system for parallel applications. In *Proceedings of the 14th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) of IPDPS*.

TESSIER, R. AND BURLESON, W. 2001. Reconfigurable computing for digital signal processing: A survey. *The J. VLSI Signal Process. 28*, 7–27.

TRIPP, J. L., MORTVEIT, H. S., HANSSON, A. A., AND GOKHALE, M. 2005. Metropolitan road traffic simulation on FPGAs. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE Computer Society, Washington, DC, 117–126.

TRUONG, H.-L. AND FAHRINGER, T. 2002. SCALEA: A performance analysis tool for distributed and parallel programs. In *Proceedings of the 8th International Europar Conference(EuroPar02)*. Springer, 41–55.

UNIVERSITY OF CALIFORNIA AT RIVERSIDE. 2010. ROCCC 2.0 user's manual—Revision 0.5.1. http://roccc.cs.ucr.edu/documentation/files/UserManual-0.5.1.pdf.

WILLIAMS, J., GEORGE, A. D., RICHARDSON, J., GOSRANI, K., MASSIE, C., AND LAM, H. 2011. Characterization of fixed and reconfigurable multi-core devices for application acceleration. *ACM Trans. Reconfig. Technol. Syst. 3,* 4, to appear.

WILLIAMS, J., GEORGE, A. D., RICHARDSON, J., GOSRANI, K., AND SURESH, S. 2008. Computational density of fixed and reconfigurable multi-core devices for application acceleration. In *Proceedings of the Reconfigurable Systems Summer Institute (RSSI)*.

XTREMEDATA INC. 2010. XD1000™ development system. http://old.xtremedatainc.com/index.php?option=com_content&view=article& id=109&Itemid=170.

Received August 2010; accepted January 2011