The DARPA Boolean equation benchmark on a reconfigurable computer

D. A. Buell and S. Akella and J. P. Davis and G. Quan
Department of Computer Science and Engineering
University of South Carolina
Columbia, South Carolina 29208
{buell|akella|jimdavis|michalsk|gquan}@cse.sc.edu
D. Caliga
SRC Computers, Inc.
4240 North Nevada Avenue
Colorado Springs, Colorado 80907
caliga@srccomp.com

Abstract

The Defense Advanced Research Projects Agency has recently released a set of six discrete mathematics benchmarks that can be used to measure the performance of high productivity computing systems. Benchmark five requires matching a short bit string (with don't care positions) against a very long bit stream, setting up systems of linear equations with $0 - 1$ coefficients, and solving the systems using Gaussian elimination. We describe the implementation of this benchmark on the SRC Computers reconfigurable computer and present results on performance. Since this is a reconfigurable machine with Field Programmable Gate Arrays (FPGAs) that can be used as processing elements, the implementation has many features of a special purpose hardware design as well as the load balancing and data access problems inherent in a software implementation.

1. Introduction

The Defense Advanced Research Projects Agency has recently released the DARPA HPCS Discrete Mathematics Benchmarks that can be used to measure the performance of high productivity computing sytems [1]. These benchmarks are intended to augment the DARPA floating point benchmarks as well as standard performance guides such as LINPACK. Described briefly, the six benchmarks (numbered zero through five by DARPA) are

0. random access to a very large shared memory array;

1. matrix multiplication with multiprecise modular coefficients;

2. a dynamic programming problem;

3. transposition of bits in a bitstream;

4. integer sorting;

5. matching of a bit string with a bitstream and solution of a derived system of linear boolean equations.

Benchmark five requires matching a short bit string (including don't care bits) against a very long bitstream, setting up systems of linear equations with $0 - 1$ coefficients, and solving the systems using Gaussian elimination. We describe an implementation of this benchmark on the SRC Computers SRC-6 reconfigurable computer [2], and we report on this implementation. We provide raw performance numbers, an analysis of the SRC-6 for this application, and a more general discussion of the issues of load balancing of data movement and computation for problems similar to this benchmark.

2. Benchmark Five

Let $\{s_i\}$ be a stream of bits of length L. We are to search for all occurrences of a bit pattern P, where bits in P can be specified as 1, 0, or "don't care" bits. Let j be the position in the bitstream immediately after an occurrence of P in the bitstream.

Beginning with bit j, we form N equations in N unknowns over $GF(2)$, solve the system of equations, and output either the unique solution or the information that no solution exists.

The systems of equations should be set up and solved for every occurrence of P in the bitstream.

mark five.

- The length L of the input bitstream is 10^7.

- An example bit pattern is the pattern of 22 bits

$$P = 000???100??110?10?1111,$$

  where $?$ indicates a "don't care" bit.

- We take $N = 700$, and thus we are required for every substring match to set up and solve the 700×700 matrix equation

$$\begin{pmatrix} s_j & s_{j+1} & \cdots & s_{j+699} \\ s_{j+701} & s_{j+702} & \cdots & s_{j+1400} \\ \cdots & & & \\ s_{j+489999} & s_{j+490000} & \cdots & s_{j+490698} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_{700} \end{pmatrix}$$

$$= \begin{pmatrix} s_{j+700} \\ s_{j+1401} \\ \cdots \\ s_{j+490699} \end{pmatrix}$$

  using the $700 \times 701 = 490700$ bits beginning with the j-th bit.

3. Software Implementation

These benchmarks are new to DARPA as benchmarks, but they have been in use for some time. However, the available software implementations are rather dated, which makes it difficult to make comparisons against prior art. The "rules of the game" for the DARPA benchmarks are that one should first make and them time a code port with only those changes necessary for correct execution, and only later make subsequent changes that would speed up the execution based on machine-specific characteristics.

However, by being old, the benchmarks present several problems that make it difficult to use the old code "as is." First, the data size for benchmark 5 is too small. A 22-bit pattern to match is not necessarily too small, but a bitstream of only ten million bits is insufficient to tax a modern computer. When ported as indicated below, the benchmark code on ten million bits of input data takes only about 0.07 second. The signal-to-noise ratio of timing something that small makes any conclusions highly suspect if one is using services provided through an operating system and not actual hardware counters for timing.

Since the benchmark as originally stated is too small, we have increased the size of the input bitstream from ten million to 1600 million bits, specifically to $((3/2) \cdot 2^{30} = 1610612736)$ bits. This is not a magic number, and we reserve the right to increase the size even more. But one further problem with the code provided for the benchmark is

manner that is fast in time but wasteful in memory. When we increase the bitstream much larger than 1.6 gigabits, the concomitant increase in array sizes elsewhere in the program cause problems. In order to maintain as much fidelity to the benchmark rules as possible, we thus stop with the size indicated here. Our execution results clearly seem to scale linearly with the number of bits processed, so we feel that extrapolating to larger bit lengths is justified based on the experimentation presented here.

To ensure that we are in fact solving the benchmark problem as posed, we have also in C a program for the benchmark for an Intel Pentium 4 processor. We have begun with a naive implementation that, although slow, was simple, so that the answers could readily be verified and we would have correct answers against which to compare when we tried running optimized code. A second step would be to write efficient C code for a standard Pentium. We have not done this, so we cannot yet test a highly optimized version in C or Fortran against the implementation on the SRC-6. However, we hope that over time there will develop a collection of results that will permit these results to be put into context. And once again, our final result scales linearly and can therefore be used for extrapolation purposes, so we feel that a useful experiment has in fact been conducted.

4. The Code Port

There were a number of serious issues in porting the original single-processor code, written in FORTRAN 77 for a Cray vector machine, to a modern Pentium-based machine. Most of the initial problems stemmed from the fact that the `INTEGER` data type in Cray FORTRAN is/was a 64-bit value, all the bit-oriented functions (shifts and such) operated on 64-bit integer values, and Cray FORTRAN had built in functions for `LEADZ` and `POPCNT` to provide the number of leading zeros in a word and the number of nonzero bits in a word. Standard C compilers on 32-bit Pentium machines permit 64-bit `long long` data types, but the compilers do not appear to generate correct code for shifts of more than 32 bits. Further, although software routines were provided to substitute for `LEADZ` and `POPCNT`, the code was incorrect. Finally, some of the static memory allocations of the FORTRAN code had to be changed to dynamic allocations in C.

After some substantial effort, we were able to run the modified code on a Dell PowerEdge 2650 machine (named `seymour`) with dual 2.8GHz processors and 2 Gbytes of memory. Since we have an all-C reimplementation of the benchmark that generates correct answers (albeit much more slowly than desired), we can verify correctness of our more optimized implementations.

Part of our goal in this project is to measure the effectiveness of the SRC MAPC compiler for generating code to run on the FPGAs of the MAP. We have at this point therefore, in an attempt to maintain the spirit of "programming" and not "hardware design," eschewed resorting to VHDL. Since our first intent is to measure the performance of the SRC-6 on the pattern matching part of the benchmark, we have excerpted that part of the benchmark. A C main program reads a bitstream and the necessary lookup tables from files, calls a MAP function for doing the pattern match, and then writes the results to a file. The times required for the pattern-match portion of the benchmark done entirely in C in software on `seymour` are presented as column A in Figure 1. For the expanded 1.6-gigabit benchmark of the last line, the time of approximately 10.15 seconds is the same as was required for the pattern match when done in Fortran as part of the larger program ported from the older Cray code. Our goal, then, is to use the MAP hardware to decrease the 10.15 seconds execution time of the two software versions.

5.1 A Code Port for the MAP

Most of the initial work to port the software version to a MAP version dealt with the problems of data movement. The original code stores everything in multiple long arrays as would be reasonable for a supercomputer with a large flat common memory. The MAP, however, has six banks each of 4 megabytes, and we must use this memory efficiently. Since we are dealing with patterns of bits that must be treated as unsigned 64-bit quantities, we will refer only to (64-bit) words; there are thus six on-board memory (OBM) banks each of 524288 words. The bitstream being $(3/2) \times 2^{30} = 1610612736$ bits, or 25165824 words, the bitstream cannot be loaded entirely into the MAP. In our Version 1 code we have blocked the bitstream into blocks of 262144 words and used DMA to transfer these blocks to the MAP in a loop controlled from the MAP.

Many of the C constructs for programming the MAP are self-explanatory. Some require only a brief parsing for a first-level understanding of their function. The `OBM_BANK_x` lines serve to declare the argument variables to have the given data type as arrays of the given length and to reside in the appropriate bank (`A` through `F`) of on-board memory. The Version 1 code also has declarations that allocate two arrays into banks `A` and `B`. The `DMA_CPU` function calls cause arrays to be transferred via DMA from common memory to on-board memory `CM2OBM` or from on-board memory to common memory `OBM2CM`. At present, some care must be taken in aligning the DMA transfers, but memory access can be improved by striping arrays into multiple banks of on-board memory (although we did not use strip-

ing here). The `wait_DMA` is the obvious barrier synchronization primitive.

With this basic program structure, the main program and MAP program are as presented in Figures 2 and 3. The timing of our routines is presented in column B of Figure 1.

The time of 1.67 seconds on the MAP is a speedup of 6.1, and we analyze the code to see how to improve this.

5.2 Improvements

The timing of this program depends on three factors. The first is the data movement of the bitstream to the MAP and the subsequent movement of the results array back to the host machine. The second is the number of clock ticks required for each iteration of the main loop of the MAP routine. Finally, as is traditional in high-performance computing, we ought to examine the possibility of overlapping the data movement with computation so as to hide the time spent accessing data.

One of the constraints on the current implementation is that we have six lookup tables, a large input bitstream, and an output bitstream, but only six banks of memory. If we access all six lookup tables in parallel, there will be inherent bank conflicts with the access of the input data and the storage of the result data. Further, we must consider the cost of data movement itself. If we are required to stream the bits into a single bank of memory, then the 25.165 million words, at one word per tick at 100MHz, will require about 0.252 seconds. The maximum speedup we could obtain, then, would be a factor of about $10.15/0.252 = 40.3$ over the software time of 10.15 seconds. In column C of Figure 1 we present the observed data movement time for the code of Figures 2 and 3 but with the computational loop removed. We note that the data movement time for the last column is about 0.659 seconds, or about 2.6 ticks per word on average. This means that our computational loop should be taking about $(1.671 - 0.659)/0.25165824 = 4.02$ clocks per iteration. Indeed, the MAP compiler has informed us that due to bank conflicts it has added two extra clocks per iteration and due to other resource conflicts it has added one more, for a total of four clocks per iteration.

5.3 Reducing Conflicts

We have two strategies to pursue in reducing the execution time. First, we need to reduce the execution time of the computational loop to one clock per iteration by removing the bank and other resource conflicts. Next, we can try to overlap data movement of "the next block" with computation on "this block" of data. If this can be accomplished, we would hope to be able to reduce the time by the 25165824 ticks necessary for moving the data since the data movement would be entirely overlapped with the computation.

flicts that will prevent a one-tick-per-iteration execution unless we are more clever than in our original code port.

1. Since the entire `S[.]` array is stored in a single bank, the simultaneous fetch in the `i`-th iteration of both `S[i]` and `S[i+1]` will require two fetches and cost us one tick per loop iteration. If we rewrite the loop so as to compute and save `t5` and `t6` in the previous iteration of the loop, this conflict disappears.

2. In our MAP code we have spread the lookup tables over the six banks of memory so we could do the six lookups simultaneously. This creates a bank conflict with reading the bitstream array `S[.]` and with writing the result array `SS[.]`, since we would like to be able read and write these arrays at the same time we are fetching lookup values from the same banks.

3. We note that we have avoided a scalar dependency by using the vendor-provided `cg_accum` function. A customary code block

```
(*sscount) = 0;
for(i = 0; i < limit; i++)
{
  code
  if(t7 != 0)
  {
    SS[*sscount] = i;
    (*sscount)++;
  }
} /* for i from 1 to num */
```

would have required one extra tick to increment the pointer and would have caused a slowdown. The use of the `cg_accum` function

```
cg_accum_add_32(1,t7!=0,0,
                ((loopsub==0)&(i==0)),
                &localsscount);
SS[localsscount] = i + inputarraysub + 1;
```

prevents the loss. This function performs a 32-bit addition into `localsscount` (the last argument) of the values in the first argument (in this case the constant 1) for every loop iteration for which `t7 != 0` is true, resetting the value of `localsscount` to zero (the third argument) every time the condition `(loopsub==0)&(i==0)` is met.

The first of these changes can be done with the existing code. The second, however, will require consolidating the lookup tables so as to use fewer of them, thus saving a bank or two for the input and the output streams. We have not made these changes, but we have run the code as if we

ure 1. Were we to make these changes we would produce a loop that executes in one tick per iteration, and the difference between the 65 million clock ticks of column C for data movement and the 90 million ticks for total execution time of column D is the cost of one tick per iteration on the 25 million words of the data stream.

5.4 Overlapping Data Movement and Computation

The final improvement that could be made in this program is to overlap the DMA of the data into the MAP with computation on the data previously brought into the MAP. When this change is done, as is shown in the Version 2 code of Figure 4, we get the execution times of column E of Figure 1. As can be seen, we have decreased the execution time by exactly the 25 million ticks we spent waiting for the DMA of the data to finish.

5.5 The Bottom Line

Although we have in fact not implemented "the fastest possible" version of code for the benchmark, we are confident that such an implementation would be entirely feasible and straightforward. The current timings for the code as implemented readily allow for execution at the 100MHz rate of the hardware. Silicon usage for the various implementations has been in the range of 13%-15% of the 33792 slices of one Xilinx Virtex 6000 chip (the MAP has two such chips).

The most crucial part of an implementation that would run at one tick per data word is reducing the number of lookup tables from six to four. There are a total of 85 bits that must be considered in order to determine whether a bit in a given 64-bit word could be the first bit of a match of the target pattern. If we were to expand our lookup from 16 bits using $2^{16} = 65536$ words per lookup table to 19 bits using all $2^{19} = 524288$ words of a memory bank, then four lookup tables would cover $4 \times 19 = 76$ bits. The remaining $85 - 76 = 9$ bits could readily be accommodated using a separate lookup table stored in the block RAM on the FPGA.

Finally, we comment on the level of effort necessary to achieve this implementation. By far the largest effort needed did not relate directly to the reconfigurable platform but rather to the problems of the algorithm itself. The original code was old and buggy, and the difficulty of verifying correctness (the built-in self test of the supplied code was comprised of hard-coded lists of matches and thus depended on the bits supplied by the random number generator) made it necessary to be very careful in checking that the code was functioning as desired.

We have shown that the DARPA benchmark 5 code can be ported to the SRC-6, and that the pattern matching sub-step that is one of the computational cores of this benchmark could be done at the rate of one word of data per clock on the SRC-6. Our future work will be to adapt the lookup tables to permit an implementation that in fact would run at this maximum speed.

We believe that this represents a watershed event in the history of computing. We have taken an extant implementation in a high level language; we have maintained a process of high-level language programming, and we have achieved the maximum possible processing rate while resorting only to the analysis of bank conflicts and loop dependencies that have been standard in vector and parallel programming for two decades. The programming process has indeed been a programming process, and the analysis has been software timing and debugging as is common with parallel programs.

The era of effective programming of a reconfigurable computer has arrived.

7. Acknowledgements

The USC authors are grateful to Esam Al-Araby, Hatim Diab, and Proshanta Saha of George Washington University for their assistance in making available the SRC-6 machine at GWU.

References

[1] Defense Advanced Research Projects Agency. High productivity computing systems discrete mathematics benchmarks, 2003.
[2] SRC Computers, Inc. Web site. www.srccomp.com.

Figure 1. Timing results

	A SW only	B MAP V1	C Data only	D No conflicts	E Overlap V2
16	0.11	0.028	0.017	0.022	
32	0.22	0.045	0.024	0.031	
64	0.44	0.079	0.037	0.049	
128	0.85	0.149	0.064	0.087	
256	1.73	0.285	0.118	0.160	
512	3.40	0.559	0.228	0.313	
768	5.12	0.837	0.334	0.457	
1024	6.79	1.108	0.440	0.609	
1536	10.15	1.671	0.659	0.901	0.256

Figure 2. Main program, Version 1

```c
#include <map.h>
#define LMAX 1610612736
#define NBPW 64
#define NMAX LMAX/NBPW+1
#define TWOTO16 65536
#define TWOTO16MINUS 65535
#define LUTLENGTH TWOTO16
#define BLOCKSIZE 262144
void pp5csub(int64_t loopcount,
  int64_t inputcountperloop,
  uint64_t Sblock[],
  uint64_t locallut0[],uint64_t locallut1[],
  uint64_t locallut2[],uint64_t locallut3[],
  uint64_t locallut4[],uint64_t locallut5[],
  uint64_t SSblock[],int64_t *sscount,
  int64_t *maptime,int mapnn);
int main()
{
  int mapnum;
  int64_t bitcount,i,inputcountperloop,loopcount,
          maptime,sscount;
  uint64_t *S,*SS;
  uint64_t *locallut0,*locallut1,*locallut2,
          *locallut3,*locallut4,*locallut5;
  // mandatory allocation of the MAP
  mapnum = 1;
  if(map_allocate(mapnum))
  {
    fprintf(stdout,"Map allocation failed.\n");
    exit (1);
  }
  MALLOC THE S, SS, AND locallut ARRAYS ON CACHE ALIGNED
  BOUNDARIES USING VENDOR-PROVIDED CALLS FOR THE MALLOC

  READ THE DATA AND LOOKUP TABLES

  inputcountperloop = BLOCKSIZE;
  loopcount = bitcount/(NBPW*inputcountperloop);

  pp5csub(loopcount,inputcountperloop,S,
          locallut0,locallut1,locallut2,
          locallut3,locallut4,locallut5,
          SS,&sscount,&maptime,0);

  // highly advised deallocation of the MAP
  if(map_free(mapnum))
  {
    fprintf (stdout,"Map deallocation failed.\n");
    exit(1);
  }
  OUTPUT THE SS RESULT ARRAY
  return(0);
}
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

fastcase®
Smarter legal research.