Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2011) 1–10

Procedia Computer Science

The Sixth International Workshop on Automatic Performance Tuning, iWAPT 2011

Autotuning a Random Walk Boolean Satisfiability Solver

Tao Cui and Franz Franchetti[1]

Department of Electrical and Computer Engineering, Carnegie Mellon University

Abstract

In this paper we present a performance optimization case study for a kernel with dynamic data structures, few instructions on boolean variables per node, and data-dependent control flow. This kernel serves as model for wide class of important algorithms operating on dynamic data structures. Our example is a simple random walk search algorithm to solve boolean satisfiability (SAT) and our goal is to gain better understanding on how such a kernel must be optimized, not to build a better SAT solver. We start from a very compact, reasonable C++ implementation using the STL vector class, and arrive at a fully optimized, parallelized, vectorized C implementation in which we use algorithm-level optimization, data structure and layout optimization, and program generation and autotuning. We evaluate our example on a 2.66 GHz quad-core Core 2 Extreme in 32-bit mode and a 3 GHz dualcore Xeon 5160 in 64-bit mode. On the Core2 Extreme quadcore processor for L1 resident synthetic benchmarks, the final version achieves up to 47% of machine peak (240 useful logical operations per cycle) and the realistic benchmarks 60–230 operations/cycle. The baseline C++ STL implementation requires between 7 and 50 cycles per useful logical operation for the synthetic benchmark; this translates into about 1,700–12,000x performance gain.

Keywords: Boolean satisfiability, program generation, automatic performance tuning
PACS: 89.20.Ff

1. Introduction

The last decades have seen an enormous growth in the performance capabilities of computing platforms. A current Intel server processor has a double precision peak performance of more than 200 Gflop/s (10^9 additions/subtractions/multiplications per second) thanks to an 8-core CPUs with AVX vector instructions. This is counting only a single CPU; including graphics accelerators and multiple CPUs would yields another 10x increase in peak performance. The enormous performance levels of current CPUs have enabled tremendous productivity gains: they enabled higher-level languages and software engineering methods that allow us to build and maintain complex software systems [1]. Almost always, a clear decision is made: productivity is chosen over efficiency of the final software product. This choice is often reinforced by the difficulty and high cost of achieving good efficiency on today's computing platforms which include deep memory hierarchies, multiple processor cores, explicit data transfer, graphic processors, and special instruction sets. Thus, a typical application today is often performing one or more orders of magnitude below the processor's capabilities.

[1]This work was supported by NSF through awards 0702386 and 0931987, by ONR through grant N000141110112, and by Intel and Mercury.

In this paper we report on a case study in full performance optimization, including algorithm-level optimization, data structure and layout optimization, and program generation and autotuning, for a simple random walk solver for boolean satisfiability (SAT). This solver is a non-numeric algorithm with highly data-dependent control flow. The underlying data structure is a doubly nested linked list, and the computational kernel performs only a few boolean operations per visited node. We start from a reasonable C++ STL (standard template library) version of the kernel and first translate it into an ANSI C baseline implementation. Next we develop an optimized ANSI C implementation, by translating dynamic data structures into arrays, translating control flow into data flow, and performing data layout optimizations and bit-vectorization. We employ program generation for code specialization and to build jump tables for unrolled while loops. Then we use SIMD instructions for wider machine words and build a multi-thread solver, and perform the necessary algorithm-level changes. Finally we employ autotuning to select the best integer kernels and tune the size of the generated jump table and a logical constant table used for vectorized bit flipping. We evaluate our approach on a quad-core 2.66 GHz Core 2 Extreme in 32-bit mode and a dualcore 3 GHz Xeon 5160 in 64-bit mode. On a Core2 Extreme quadcore processor and for L1 resident synthetic benchmarks, the final version achieves about 47% of machine peak (240 useful logical operations per cycle) for a synthetic benchmark run; for realistic SAT problems the performance varies between 60 and 230 useful logical operations per cycle. The baseline C++ STL implementation required between 7 and 50 cycles per useful logical operation for the synthetic case. This translates into about 1,700–12,000x performance gain.

Contribution. The contribution of this paper is a case study investigating the techniques necessary to achieve high efficiency for a class of non-numeric data-dependent kernels on current multicore processors with SIMD vector extensions. The methods applied throughout the paper are applicable to the whole algorithm class and the presented case study is a first step towards automating the performance optimization of these algorithms. Specifically, we investigated the following aggressive optimization techniques of hand-tuners that are beyond the reach of compilers:

- Global data structure reorganization and data recoding from linked lists into arrays.
- Conversion of conditional assignments into data flow and unrolling of uncountable loops.
- Multi-threading and bit-level SIMD vectorization requiring algorithmic optimization.
- Program generation for code specialization and autotuning of free parameters.

The final result is a kernel that performs at about 50% of the multicore processor's peak performance for boolean operations. To the best of our knowledge, the type of algorithm, data structures, and optimization techniques underlying our paper have not been addressed in the autotuning community at that performance level. While we use a simple SAT solver as example, we do not claim to provide a new and improved SAT solver for real-world problems.

Related Work. *Program generation* (also called generative programming) has gained considerable interest in recent years [2] The basic goal is to reduce the development, maintenance, and analysis of software. The Rose framework [3] is a source-to-source compiler that is capable of optimizing complicated C++ code and to remove the penalty of higher abstraction. CPU vendors like IBM, Intel, and AMD provide highly hand-optimized performance libraries for their systems (ESSL, MKL, and ACML, respectively). GotoBLAS is a highly hand-tuned version of the BLAS library. Automating the optimization of performance libraries is the goal in recent research efforts on *automatic performance tuning* and adaptive library frameworks that can offer high performance with greatly reduced development time. Examples include ATLAS [4], Bebop/Sparsity [5], FFTW [6], and Spiral [7].

2. Boolean Satisfiability (SAT)

SAT. Boolean Satisfiability (SAT) is the canonical NP-complete problem: given a propositional formula, determine whether there exists a variable assignment such that the formula evaluates to *true* [8]. As example, consider the following formula in the conjugate normal form (CNF):

$$f(x_0, \dots, x_8) = (x_0 + x_3' + x_5')(x_1 + x_2' + x_6 + x_7)(x_3' + x_4')(x_3 + x_5' + x_7 + x_8'),$$

where multiplication denotes *logical and*, addition *logical or*, and $(.)'$ *logical negation* (`&`, `|`, and `~` in C syntax, respectively). x_0 is called a variable, x_3' a literal, and $(x_0 + x_3' + x_5')$ a clause. A binary vector x that assigns each x_i

```
// CNF data is stored in STL vector<>
class literal_c { bool negate; int var; };
class clause_c { vector<literal_c> clause; };
class sat_c { vector<clause_c> cnf;
  vector<bool> variables;
  bool EvalCNF(); bool RandomWalk(); };

// evaluate a CNF for a variable vector
bool sat_c::EvalCNF() {
  bool res_cnf = true;
  for(int i=0; (i<cnf.size()) && res_cnf; i++) {
    bool res_clause = false;
    for(int j=0; j<cnf[i].clause.size(); j++) {
      bool val = variables[cnf[i].clause[j].var];
      res_clause = res_clause ||
        cnf[i].clause[j].negate ? !val : val;
    }
    res_cnf = res_cnf && res_clause;
  }
  return res_cnf;
}

// Random walk search
bool sat_c::RandomWalk() {
  bool result = false;
  do {
    variables[rand() % variables.size()].flip();
    result = EvalCNF();
  } while(!result);
  return result;
}
```

```
// CNF data is stored in nested linked lists
typedef struct { literal_t *next;
  unsigned var, negate; } literal_t;
typedef struct { claus_t *next;
  literal_t literal; } clause_t;
typedef struct { clause_t *cnf;
  int *variables; int var_num) sat_t;

// evaluate a CNF for a variable vector
int EvalCNF(int *variables, clause_t *clause) {
  int res = 1;
  do {
    int tmp = 0;
    literal_t literal = clause->literal;
    do {
      int val = variables[literal->var];
      tmp |= literal->negate ? ~val : val;
    } while (literal = literal->next);
  } while ((res&=tmp) && (clause=clause->next));
  return res;
}

// Random walk search
int RandomWalk(sat_t *sat) {
  int result = 0;
  do {
    sat->variables[rand()%sat->var_num] ^= 1;
    result = EvalCNF(sat->variables, sat->cnf);
  } while(!result);
  return result;
}
```

Figure 1: A simple random walk SAT solver. *Left:* C++ STL version. *Right:* Linked list-based ANSI C version.

such that the formula is *satisfied* (i. e., values for $x_0, \ldots, x_8$ such that $f(x_0, \ldots, x_8)$ is *true*) is called a solution to the SAT problem.

SAT solvers. Boolean Satisfiability is a well-studied area with a vast body of work. Algorithms are based on various principles such as resolution, search, local search and random walk, Binary Decision Diagrams, Stälmarck's algorithm, and others [8, 9, 10]. Some algorithms are complete (they can prove that there exists no solution) while others are stochastic. Depending on the application and the properties of the SAT instances one is interested in, one can choose among many algorithmic variants. These variants may be tweaked for important cases and make assumptions or require a priori knowledge. When analyzing the state-of-the-art implementations that were submitted to the SAT competition[2] we find that while solvers employ sophisticated algorithms and heuristics, they do not utilize the machine efficiently and rarely are parallelized or use SIMD vector instructions. In contrast, our toy random walk SAT solver runs highly efficient but would not be able to compete in the SAT competition due to its simple approach. However, it could provide highly efficient local search on small sub-formulas to be used as algorithmic building block in realistic solvers.

Random search. A simple algorithm to find a solution to a SAT problem is random walk search. Starting from a random assignment vector for (x_i), the formula in CNF is repeatedly evaluated, and if it is not satisfied a randomly chosen variable's value is negated. This process is repeated until either a solution is found or timeout is reached. This simple algorithm quickly finds a solution if many solutions exist. We base our discussion on this simple algorithm to focus on the performance-relevant optimization methods. Advanced versions of this algorithm do not flip a random variable, but employ heuristics and machine learning to decide which variable to flip next and cache the evaluation state of clauses. A software package implementing a highly evolved state-of-the-art version of random search is WalkSAT[3] [10].

C++ STL implementation. Figure 1 (left) shows the implementation of our simple random walk SAT solver using the C++ STL vector template. We define a class for literals, and then define a clause as a vector of literals and a CNF as vector of clauses, using the C++ STL vector template. A SAT problem is a CNF plus a variable vector. We

[2]`http://www.satcompetition.org/`
[3]`http://www.cs.rochester.edu/~kautz/walksat/`

provide a method that evaluates a CNF with a current variable vector, and a random search method that evaluates the CNF and flips a random variable, until a solution is found. Practical implementations would also contain a timeout. Using the STL vector templates, the algorithm can be implemented concisely, and the STL takes care of variable-length dynamic data structures (clauses and the full formula). However, using the convenience of the STL comes at a performance penalty.

ANSI C implementation. In Figure 1 (right) we show a C base-line implementation of the random walk SAT solver. C++ classes have been replaced by C structs and C++ STL vectors by NULL-terminated linked lists. Thus, the literals of a CNF are stored in a doubly nested linked list with a struct as data in the node, and the evaluation of a CNF becomes pointer chasing in a nested while loop, with one conditional and one unconditional logic operation in the body of the innermost loop. This C implementation can be considered a reasonable portable low level implementation.

3. Optimizing the SAT Kernel

In this section we discuss the optimization techniques necessary to achieve high machine efficiency for the given kernel. We first optimize data representation, convert unpredictable control into data flow, and use a jump table to unroll while loops. Then we exploit parallelism by performing multiple random searches in parallel (one search per bit in the data type) by vectorizing on the bit level, and by running one independent search per thread on multicore CPUs. We apply optimization techniques in sequence and note tunable parameters. Some techniques require program generation to automate tedious code duplication and specialization. During the optimization process, the concise and readable code of Figure 1 will become progressively unreadable and obfuscated, but more efficient. Autotuning will be the last step in this sequence of optimizations.

3.1. Data Structure Optimization

While from an performance perspective the C version is already an improvement over the C++ STL version, there are two problems: 1) literals are stored as structs necessitating index computation and multiple data accesses, and 2) the multiple traversals of doubly linked lists with such a small kernel per list node should be avoided if at all possible. We now discuss how to convert the whole data structure into two integer arrays.

The optimized function `EvalCNF` is shown in Figure 2. We use pointers to the current literal (`*lit`) and the current clause length (`*clauses`) to control the loops and access the CNF data. We use compact C syntax with the conditional operator `?:`, assignment operators `&=`, `|=`, postfix increment operator `++`, and pointer dereferencing `*lit` to describe all the pointer updates and computation, and to aid the compiler.

Converting the struct into an integer. Assuming that the number of variables is not requiring all bits of an unsigned integer, we can combine the boolean variable for negation and the unsigned integer variable number into a signed integer. The sign signifies whether or not the variable needs to be negated, and the absolute value gives the variable number. Current (x86) CPUs have the necessary instructions to handle signed and unsigned integers well, so checking the sign and extracting the absolute value is computationally cheap. However, since $0 = -0$, we need to shift the variable ID by at least one to enable the negation of x_0.

Converting linked lists into an array. During the random walk search the CNF does not change and the trip-count of the search loop is expected to be high if actual search is necessary. Thus, we perform an one-time conversion of the dynamic data structure into two arrays. In the conversion, we traverse the whole CNF from beginning to end. For each clause we note its length in an integer array, so that entry i contains the length of the ith clause in the formula. In a second array we collect all the literals of the formula in the order they appear. Thus, we drop the explicit clause boundary information; it can be reconstructed by consulting the clause length array. We traverse the whole data structure first to calculate its aggregate size and then allocate one block per array, removing any dynamic data management from the inner loop.

3.2. Control Flow Optimization

Optimization of conditional assignments. The conditional operator `?:` in the loop body can introduce inefficiencies in multiple ways. There is a data-dependent condition in the critical path, and depending on the CPU and compiler it will be translated into a conditional move, a data-dependent unpredictable branch, or a pair of predicated instructions. Even in the best case there is a data dependency which may stall an out-of-order processor while in the worst case an unpredictable branch stalls the processor for 10s of cycles.

If the machine supports predication or conditional moves, the original code may be best. However, if neither is supported (well) but logical operations are cheap, a different approach may be beneficial: We can store a signed integer in one's complement (as opposed to usual two's complement) with the sign in bit 0 (least significant bit LSB) and the absolute value from bit 1 on. Extracting the sign bit is done via *binary and*, while extracting the absolute value is done via right-shift by one bit. The conditional negation `l>0?var[l]:~var[-l]` becomes `(var[l>>1)^(l&0x1)` using *exclusive or* `^`.

Yet another option to use the one's complement format with the sign bit in the LSB is to store both the original value and the negated value for each variable consecutively in an array twice the size (the even locations store the values of the variables while the odd locations store the negated values), and use one's complement variable ID value directly as index into this array. Moreover, when the search algorithm flips a bit, it now also needs to flip the negated version as well. As further optimization we can simply use the memory address of the variable (the negated or un-negated version) as variable ID, and the negation is done implicitly by treating the variable ID as integer pointer and dereferencing it, saving a few more instructions. This method increases the working set size for lowering the instruction count. Which of these versions is best depends on the machine and compiler and we use autotuning to select the best solution for a given CPU/compiler combination. We show the code variants in Figure 2.

```
#define COND_NEG(l, var) (l>0?var[l]:~var[-l])
#define COND_NEG(l, var) (var[l>>1]^(l&0x1))
#define COND_NEG(l, var) (*l)

int EvalCNF(int size, int *lit,
            int *clauses, int *var) {
  int res = 1, clause_len;
  while (res && (clause_len = *(clauses++))) {
    int tmp = 0;
    for (i=0; i<clause_len; i++, lit++)
      tmp |= COND_NEG(*lit, var)
    res &= tmp;
  }
  return res;
}
```

Figure 2: Integer array-based optimized C implementation of `EvalCNF` with three kernel variants to conditionally negate a literal's value. Note that the variable ID is differently encoded across the three cases.

Unrolling the while loop. The two nested uncountable loops with only a few operations in the body result in unpredictable branches and destroy instruction-level parallelism, and also hamper register allocation and instruction scheduling. When converting the CNF to the array representation, the inner loop has been converted into a countable loop with an trip count that is depending on the outer loop and stored in a table. Still, this loops cannot be unrolled as usually done since consecutive clause have random length and may be short. However, the loop can be unrolled using a jump table, a trick that is often employed by assembly programmers and can be implemented using a switch statement. Figure 3 shows 4-fold unrolling of the inner loop using a jump table. The unrolled version has a switch statement containing a special case for all clause lengths up to 4, and for longer clauses it repeatedly peels of 4 iterations using the largest special case. We do not reuse code across the cases to enable better register allocation and instruction scheduling, at the expense of considerable code size increase. In practice, unrolling by 16 or 32 might suffice. However, if short clauses dominate it is necessary to unroll the outer loop as well, producing a jump table for multiple consecutive clauses. For instance, unrolling the outer loop by 2 we produce a jump table for all pairs of clauses of up to length 8, resulting in 256 cases. A clause of length 2 followed by length 3 would be executed by the following code fragment.

```
while (res&&(clause_len=*(clauses++))) {
  int tmp = 0;
  do {
    int clen = clause_len<4?clause_len:3;
    switch(clen & 0x3) {
      case 0x0: break;
      case 0x1: tmp |= **lit;
                lit++; break;
      case 0x2: tmp |= **lit | **(lit+1);
                lit+=2; break;
      case 0x3: tmp |= **lit | **(lit+1) |
                         **(lit+2);
                lit+=3; break;
    }
  } while (clause_len-=clen);
  res &= tmp;
}
```

Figure 3: Unrolling the inner while loop by 4, including code that peels iterations for more than 4 iterations.

```
case 0x32: tmp = (**lit | **(lit+1)) & (**(lit+2) | **(lit+3) | **(lit+4)); lit+=5; break;
```

In addition, more complicated loop peeling that handles clauses that are longer than the maximum unrolled clause length is needed. Given the highly regular and repetitive nature of these code fragments we employ a small program generator to automatically generate the switch statement. Many influences (instruction cache size, CNF statistics, relative cost of loads vs. computation) determine how large the switch statement has to be; we employ autotuning to find the best size.

3.3. Parallelization and Vectorization

Next we discuss parallelization and vectorization. Our SAT solver performs a search that evaluates the same CNF for many independent variable vectors. Evaluation of CNFs naturally operates on bits and machine words are wide (32 or 64 bit integers are usual). Thus we can evaluate multiple variable vectors simultaneously, by assigning each vector a different bit plane (the same bit across multiple variables), performing bit-level vectorization or SIMD within a register (SWAR) [11]. This approach requires an adaptation of the bit flipping mechanism. Similarly, evaluating the CNF for multiple variable vectors can be done in parallel on multiple processor cores with minimal need for synchronization.

Bit-Vectorization. The native data type of logical variables is a single bit, however, a standard C implementation uses a machine word (integer) to represent bits. The key algorithmic idea is to evaluate the CNF for *n* variable bit-vectors simultaneously, vectorizing the search loop by packing *n* bits into an *n*-bit register, storing each variable bit-vector in a bit plane. Standard C integer data types allow for 8-way to 64-way vectorization using data types `char`, `short`, `int`, and `long` or compiler-specific fixed-bit width integer data types (`__int8`, to `__int64`, respectively). It is typically most efficient to work with the native word size, since both too wide and to narrow words induce overhead. By going beyond standard C and utilizing special SIMD instruction sets, even wider vectors are possible. For instance, Intel's vector ISAs provide 64–512 bit vector registers (MMX, SSE, AVX, and Larrabee).

If bit vectorization is used with standard C data types, the CNF evaluation code does not change at all since our code already uses the correct binary and logical C operators to perform bitwise logic and to check if a result is found in any bit plane. For SIMD instruction sets we have to use the intrinsic function interface that abstracts vector operations and vector registers. The variables have to be declared as SIMD vectors (interpreted as bit-vectors) and the logic and binary operations have to be translated into their respective SIMD instructions, using intrinsic functions. For instance, for the 128-bit SSE family, a logical and operating on the 128-bit XMM registers is performed using the data type `__m128i` and intrinsic functions `_mm_and_si128` and `_mm_or_si128`. In the case of SSE, checking if any bit in a SIMD vector is set requires different instruction sequences for 32-and 64-bit mode up to SSSE3 while SSE4.1 introduces the `_mm_testz_si128` instruction that that performs the needed check, as shown below.

```
// SSE 4.1
#define  is_nonzero(xmmv) _mm_testz_si128(xmmv, xmmv)

// SSE2/EM64T
#define is_nonzero(xmmv) (_mm_cvtsi128_si64(_mm_or_si128(xmmv,_mm_srli_si128(xmmv, 8)))!=0)

// SSE2/32-bit x86
__forceinline __int32  is_nonzero(xmmv) is_nonzero(__m128i xmmv) {
  __m128i xmmv8 = _mm_or_si128(xmmv, _mm_srli_si128(xmmv, 8));
  return _mm_cvtsi128_si32(_mm_or_si128(xmmv8, _mm_srli_si128(xmmv8,4))) != 0;
}
```

Below we one line of the SIMD version of the jump table in Figure 3. The line evaluates a clause of length 2 followed by a clause of length 3, using the 128-bit SSE data types, performing the operation for 128 searches in parallel.

```
case 0x32: tmp = _mm_and_si128(_mm_or_si128(**lit, **(lit+1)) _mm_or_si128(**(lit+2),
                               _mm_or_si128(**(lit+3), **(lit+4))));
           lit += 5; break;
```

While evaluating the CNF can be accomplished with the same code whether a single bit or a bit-vector is stored in a register, the flipping of random bits during the search requires attention. Since *n* searches are performed in parallel, *n* random bit variables need to be flipped since each search happens on in a different bit layer of the integer variable array. Figure 4 (left) shows ANSI C code that flips one random bit variable per bit layer, using the C unsigned integer data type. The flipping overhead increases linearly with the bit-width of the data type while the vectorized evaluation of the CNF for *n* bit vectors is done in the same cost as for a single bit vector stored as boolean value in an integer register. As optimization, we flip the same variable in multiple searches (bit planes), performing exclusive-or operations with bit masks containing more than one set bit. This entangles multiple searches and is potentially biasing the random walk. To achieve good random search quality, we need to make sure that exactly one bit is flipped for every bit-vector (in one bit plane) and that across multiple bit flips in the search loop, the entangled planes change

```
int RandomWalk(sat_t *sat) {
  int result = 0;
  do {
    unsigned mask = 1;
    do {
      sat->variables[rand()%sat->var_num]^=mask;
    } while(mask<<=1);
    result = EvalCNF(sat->variables, sat->cnf);
  } while(!result);
  return result;
}
```

```
const unsigned masktab[4][4] =
  {{0x000F000F,0x00F000F0,0x0F000F00,0x00F000F000},
   {0x05050505,0x0A0A0A0A,0x50505050,0xA0A0A0A0},
   {0x00330033,0x00CC00CC,0x33003300,0xCC00CC00},
   {0x11111111,0x22222222,0x44444444,0x88888888}};

int RandomWalk(sat_t *sat) {
  int result = 0;
  do {
    unsigned *mask = masktab[rand()%4];
    for (int i=0, i<4; i++)
      sat->variables[rand()%sat->var_num]^=mask[i];
    result = EvalCNF(sat->variables, sat->cnf);
  } while(!result);
  return result;
}
```

Figure 4: *Left:* Flipping one variable per bit plane storing a different bit vector. *Right:* Flipping 8 bit different planes in 4 variables to flip one bit in each of the 32 bit-planes bit plane using bit patterns. Randomly chosen mask-table makes sure that always 32 bits are flipped, but different planes are entangled across flips.

randomly. To accomplish that, we build a table of multiple groups of flip patterns. Each group contains one flip pattern per integer variable that will be flipped, and within one group each bit plane gets flipped once. Across groups bit planes are assigned to different pattern numbers. The group is chosen randomly for each flip operation.

In Figure 4 (right) we show an example where eight bits are flipped by every flip-mask and each group contains 4 masks, cumulatively flipping one bit in each of the 32 bit-planes. A bit flip operation first chooses a group number, and then four variable numbers at which bits will be flipped. At each of this 4 locations an exclusive-or operation of the current integer value with the respective flip-mask is performed to flip 8 bits. Together this flips one bit in each of the 32 bit-layers of unsigned int. In the final fully optimized version we use a large table with more irregular patterns to achieve good pseudo-randomness. Moreover, we compute multiple pseudo-random numbers using a shift-register implemented with SIMD instructions, and inline this computation to achieve better performance for the bit-flipping operation. The number of bits flipped per integer variable and the size of the flip-table are tunable parameter.

Multithreading. Multicore CPUs require thread-level parallelization to be fully utilized. Since the random walk SAT solver performs a brute force search, parallelization for shared memory using multiple threads is straightforward. Each thread performs a separate search using a private vector of variable values, and the threads share the CNF data structures since CNFs are read-only. Threads need to check periodically if another thread has found a solution so they need to abort while they should signal immediately if they found a solution. This approach can be easily and efficiently implemented using the pthreads library to launch one worker thread per core. Further, we use a shared integer variable declared `volatile` for low-cost synchronization. If a thread finds a solution it atomically sets a bit corresponding to its ID in the shared variable using the intrinsic functions `_InterlockedExchangeAdd` mapping to the instruction `lock xadd`. Workers periodically poll this shared variable and abort if it becomes non-zero. As optimization, worker threads are kept alive between searches to minimize thread launching overhead. Using this approach, almost linear speed-up is achievable for cache-resident problem sizes. We show our multi-threaded random walk search in Figure 5.

```
volatile int result = 0;

unsigned *RandomWalk_th(sat_t *sat, int tid) {
  int result_th = 0;
  unsigned *var_th =
    malloc(sat->var_num*sizeof(unsigned));
  memcopy(var_th, sat->variables,
          sat->var_num*sizeof(unsigned));
  do {
    var_th[rand()%sat->var_num] ^= 1;
    result_th = EvalCNF(var_th, sat->cnf);
  } while(!(result_th || result))
  InterlockedExchangeAdd(&result,
    result_th<<tid);
  return var_th;
}
```

Figure 5: Worker function evaluating a CNF on a private variable vector, signaling a solved problem with its team members. One copy of `RandomWalk_th` runs on each core.

3.4. Autotuning

In the optimization discussion throughout this section we identified parameters that are machine-dependent and thus lend themselves to autotuning. In our example, the parameter space itself is not big and can be evaluated ex-

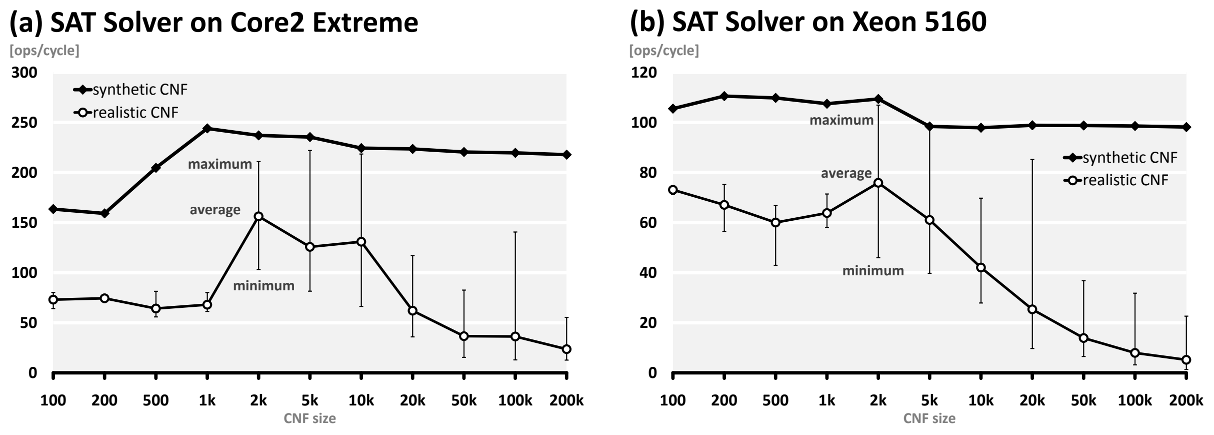


Figure 6: Performance evaluation of the random walk SAT solver for a synthetic (best case) benchmark and a realistic benchmark. (a) Performance on 2.66 GHz Core2 Extreme (quad-core). (b) Performance on 3 GHz Xeon 5160 (dualcore).

haustively at installation time. However, the parameter choices lead to very different code that cannot be handled solely with the C preprocessor macros. We needed to implement a small code generator to automatically synthesize these code parts according to the value of the tunable parameter. Moreover, some of the tuning parameters are data dependent and should eventually be adjusted at runtime, performing on-line tuning. This will be future work; for now we assume a fixed data statistics known at tuning time. Below we discuss the three tunable parameters in more detail; their performance impact will be evaluated in Section 4.

Integer kernel. We discussed three possible kernels for performing the conditional negation. The differences across versions are both the computational kernel and the variable encoding. We are employing the C preprocessor to combine the correct code snippets for translating the linked lists into the necessary array format, retrieving the result, and performing the actual operation. The best choice for a given machine depends on the relative cost of loads versus logical operations, cost of mispredicted branches versus predication, and the cache size.

Unrolling the clause evaluation. A crucial performance optimization is the unrolling of the clause evaluation (while) loop. The unrolling involves the generation of a big switch statement that for each case contains specialized code evaluating one or multiple consecutive clauses of given length; the switch statement enumerates all possible combinations up to a certain upper bound. This bound and the number of clauses to be evaluated in straight-line code by each case are tunable parameters that depend on the instruction cache size of the target machine, the expected clause length distribution, and microarchitectural details (pipeline, registers, out-of-order/in order).

Bit flipping. The overhead of bit flipping for bit-vectorized code becomes non-negligible given the wide word-sizes of microprocessors (up to 64 bit), and the even wider bit-widths of SIMD vector registers (up to 512 bits). The amount of flipping that is tolerable from a performance perspective depends on the cost of flipping and the available table space (both machine dependent), and the relative frequency with which flipping occurs (input data-dependent). On the other hand, flipping too many bit planes at the same location (i.e., a too small flip mask group) or too little randomness in the entanglement (too few groups) will degrade the power of random search. Thus, we make the size of the flip mask table and the size of each flip mask group tunable parameters. Eventually, they should become online-adaptive parameters, taking into account the recent history.

4. Experimental Results

We now evaluate the efficiency of our random walk SAT solver, and quantify the contribution of the individual optimization steps as well as the autotuning parameters. We evaluate our example on a 2.66 GHz quad-core Core 2 Extreme in 32-bit mode on Windows, and on a 3 GHz dualcore Xeon 5160 in 64-bit mode on Linux. We use the Intel C++ compiler 11.0 and the SSSE3 instruction set. In this paper, we do not compare against state-of-the-art SAT solvers, since such no fair comparison can be made at this point.

Performance of full SAT problems. We first evaluate the efficiency of our fully optimized solver on random SAT problems with varying CNF sizes. We measure a CNF size in the number of literals, and choose a constant number of variables that allows for a high number of solutions. We then group literals into clauses within certain length bounds and randomly negate about half of the literals. We run two experiments: 1) a benchmark experiment in which synthetic CNFs are constructed such that all clauses are true except for the last, forcing the solver to evaluate the whole CNF every time, and 2) purely random CNFs that may or may not be solvable and where the solver may terminate the evaluation at any location in the CNF. Figure 6 (a) and (b) show the performance of the benchmark SAT instances and the average, minimum, and maximum for the realistic problem instances across the two machines. The realistic instances are much slower and have considerable variance, compared to the benchmark performance. After a ramp-up we see highest performance for L1 cache resident sizes, and a performance drop for L2 sizes. The overhead of early termination and bit flipping and the randomness of the CNFs is responsible for the large variance of performance and the drop. For small sizes the single-core version is faster that the multi-core version. On the Core2 Extreme after the initial ramp-up the benchmark instances run at 150–250 operations/cycle in L2 while the realistic instances run between 60 and 230 operations/cycle in L1; performance decreases to 20–150 operations/cycle in L2. In the best case, the C++ STL version runs at 0.02–0.15 operations/cycle; thus it is not shown. The performance on the dualcore Xeon 5160 is qualitatively the same but absolutely about half due to half the core count, benefitting slightly from 64-bit mode and faster synchronization.

Impact of optimization methods. Next we evaluate the impact of our optimization methods on the CNF evaluation in the best case scenario. We look at the CNF evaluation in isolation and set up a CNFs that is L1 resident, all and force the CNF evaluation to always run to the end. We vary the clause lengths between a minimum and maximum length, and evaluate multiple cases. We turn on optimization methods in sequence and evaluate the resulting performance on the Core2 Extreme. Figure 7 first shows the performance of the base line and 3 portable sequential optimization steps: 1) C++ STL version, 2) C linked list version, 3) fully optimized scalar version, 4) bit-vectorization in 32-bit integer registers. The cumulative gain is between 140x (from 7 cycles per operation to 20 operations per cycle) for large clause sizes and 600x (from 50 cycles per operation to 12 operations per cycle) for small clauses. Next Figure 6 (b) shows almost linear performance gain from switching to 128-bit SSE and running a multi-threaded search on 4 cores, yielding between 200–300 useful logical operations per cycle. This translates into about 1,700–12,000x performance gain for the SIMD vectorized, multi-threaded version over the C++ STL version. The performance here is substantially higher than in Figure 6 (a) since we now evaluate the best case scenario without overhead introduced by the full random search.

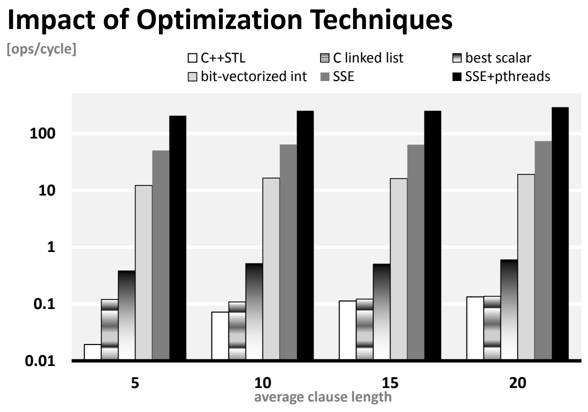
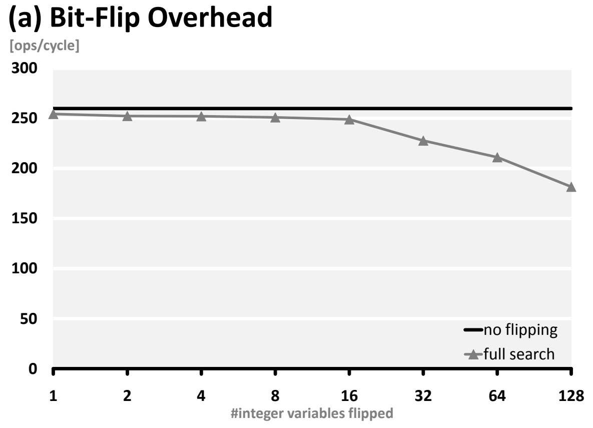


Figure 7: Impact of the various performance optimization techniques, measured on a 2.66 GHz Core2 Extreme quadcore.

Autotuning the flip mask group size. Now we evaluate the impact of vectorized bit flipping on the runtime, and tune the size of the bit mask groups. The table size is experimentally selected to be big enough and constant. We use the 128-bit single core SSE code version on the Core2 Extreme and the synthetic best case CNFs. In Figure 8 (a) we vary the number of bit masks in a group. We see that up to 16 different SSE variables can be flipped at almost constant high performance, but after that the performance degrades significantly, up to 30%. The higher numbers provide better randomness in the search, so 16 is the best choice.

Autotuning the jump table size. As last experiment we evaluate the impact of the jump table size on performance. We again pick a CNF that is L1 resident and is forcing evaluation to the end. We now vary the bounds of clause sizes and the upper bound for unrolling. Figure 8 (b) shows the performance for loop code, single unrolled clauses and pairwise unrolled clauses. We run two experiments, with clause lengths vary between 1–15 and 1–30, respectively, and vary unrolling from 2 to 16. Loop code performs better for larger clauses, but is outperformed by up to 2 times by unrolled code. Pairwise unrolling boosts performance over single clause unrolling for smaller clauses and less unrolling. Large clause length variability requires more unrolling. Thus, it is necessary to aggressively unroll for pairs of clauses without overflowing the instruction cache.



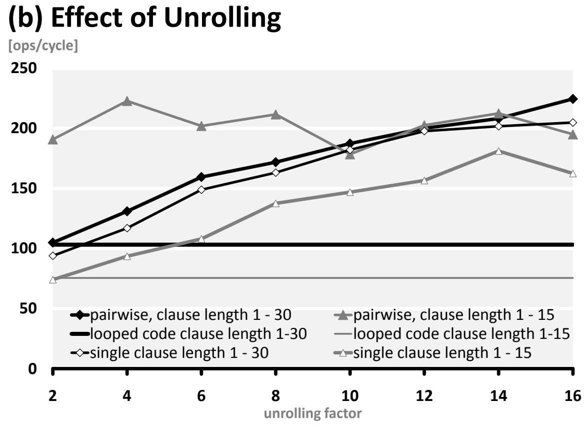


Figure 8: Performance impact of autotuning parameters on a 2.66 GHz Core2 Extreme. (a) Overhead of flipping bits in groups. (b) performance impact of loop unrolling.

5. Conclusion

In this paper we study the performance optimization necessary to derive a highly efficient random walk SAT solver, implementing a simple brute force search algorithm. This particular solver is a model for non-numerical kernels operating on dynamic data structures where the computation per node requires only a few instructions and the native data type is smaller than the machine word. We show the global program transformations that are necessary to achieve good performance, and extract tuning parameters for which we empirically find good values. The final result is a solver that runs at about 50% machine peak across multiple machines, performing 240 useful boolean operations per cycle. Reasonable C and C++ base line implementations essentially performing the same computation were three to four orders of magnitude slower. The example shows the inherent difficulty of optimizing such kernels, requiring algorithm knowledge and resulting in unmaintainable code. The high efficiency we achieve makes it possible that traditional SAT solvers could be augmented by such a kernel and would benefit from brute force local search.

The random walk SAT solver is meant as illustrative example to investigate the methods that are necessary to optimize a class of computational kernels that are traditionally not targeted by autotuning and program generation approaches and are beyond the capabilities of advanced compiler technology; it is not meant as contribution to the SAT literature. This is a first step towards automating the performance optimization of such algorithms. We regualrized dynamic data structures into streaming memory access, converted control flow into data flow, leveraged SIMD vector extensions, parallelized across multiple cores. In future work we plan to investigate our optimization approach on manycore architectures like Intel's MIC and SCC, as well as on Nvidia and GPUs and AMD APUs.

[1] J. Larus, Spending Moore's dividend, Commun. ACM 52 (2009) 62–69.
[2] GPCE, ACM conference on generative programming and component engineering.
[3] K. Davis, D. J. Quinlan, Rose: An optimizing transformation system for c++ array-class libraries, in: ECOOP Workshops, 1998, pp. 452–453.
[4] R. C. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software (ATLAS), in: Proc. Supercomputing, 1998.
[5] E.-J. Im, K. Yelick, R. Vuduc, Sparsity: Optimization framework for sparse matrix kernels, Int'l J. High Performance Computing Applications 18 (1).
[6] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93 (2) (2005) 216–231.
[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93 (2) (2005) 232– 275.
[8] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Communications of the ACM 5.
[9] L. Zhang, S. Malik, The quest for efficient Boolean Satisfiability solvers, in: Proceedings of The 14th International Conference on Computer Aided Verification, Vol. 2404 of LNCS, Springer, 2002, pp. 17–36.
[10] B. Selman, H. Kautz, B. Cohen, Local search strategies for Satisfiability testing, in: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Vol. 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1996.
[11] R. J. Fisher, A. J. Fisher, H. G. Dietz, Compiling for simd within a register, in: 11th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC98, Springer Verlag, Chapel Hill, 1998, pp. 290–304.