Cache-Optimal Algorithms for Option Pricing

John E. Savage
Brown University, Providence, Rhode Island 02912
and
Mohammad Zubair
Old Dominion University, Norfolk, Virginia 23529

Today computers have several levels of memory hierarchy. To obtain good performance on these processors it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy. In this paper, we study the computation of option pricing using the binomial and trinomial models on processors with a multilevel memory hierarchy. We derive lower bounds on memory traffic between different levels of hierarchy for these two models. We also develop algorithms for the binomial and trinomial models that have near-optimal memory traffic between levels. We have implemented these algorithms on an UltraSparc IIIi processor with a 4-level of memory hierarchy and demonstrated that our algorithms outperform algorithms without cache blocking by a factor of up to 5 and operate at 70% of peak performance.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Efficiency, Algorithm design and analysis

General Terms: Algorithms,Performance,Experimentation

Additional Key Words and Phrases: Memory Hierarchy,Cache Blocking

1. INTRODUCTION

An option contract is a financial instrument that gives the right to its holder to buy or sell a financial asset at a specified price referred to as strike price, on or before the expiration date. The current asset price, volatility of the asset, strike price, expiration time, and prevailing risk-free interest rate determine the value of an option. Binomial and trinomial option valuation are two popular approaches that value an option using a discrete time model [Kwok 1998; Cox et al. 1979]. The binomial option pricing computation is modeled by the directed acyclic pyramid graph $G_{biop}^{(n)}$ with depth n and $n+1$ leaves shown in Figure 1. Here the expiration time is divided into n intervals (defined by $n+1$ endpoints), the root is at the present time, and the leaves are at expiration times. We use $G_{biop}^{(n)}$ to determine the price of an option at the root node iteratively, starting from the leaf nodes.

The trinomial model improves over the binomial model in terms of accuracy and reliability [Kwok 1998]. The trinomial option pricing computation is represented

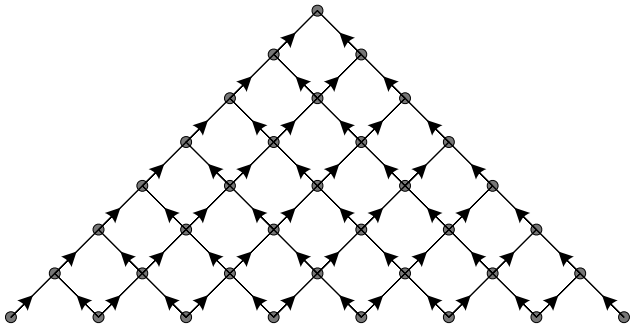
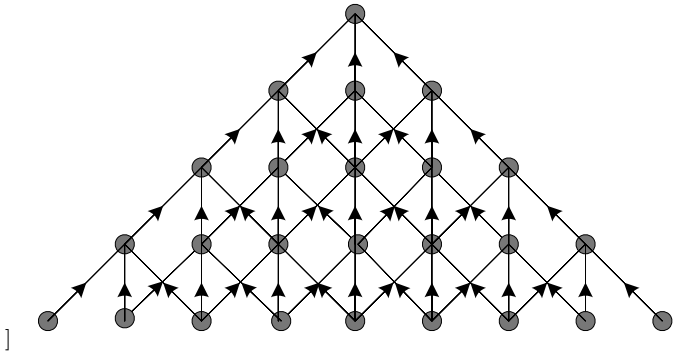


Fig. 1. The graph $G_{biop}^{(n)}$ with depth n and $n+1=8$ leaves.



]

Fig. 2. $G_{triop}^{(n)}$ with depth $n=4$ and $2n+1=9$ leaves.

using the directed acyclic graph with in-degree 3 denoted $G_{triop}^{(n)}$ of depth n on $2n+1$ leaves shown in Figure 2. As in the binomial model, we divide the time to expiration into n intervals and let the root be at the present time and the leaves be at expiration times. As in the binomial model, we use $G_{triop}^{(n)}$ to determine the price of an option at the root node iteratively, starting from the leaf nodes. The trinomial model assumes that the price of an asset can go three ways: up, down, and remain unchanged. This is in contrast to the binomial model where the price can only go two ways: up and down.

The number of floating point operations for pricing an option in both models with n intervals is of the order of n^2. We compute a better estimate of the option price with increasing value of n [Kwok 1998]. [Gerbessiotis 2004; Higham 2002; Thulasiram and Bondarenko 2002] have proposed sequential and parallel implementations for binomial option pricing that suffer from low performance. This occurs because they do not address memory hierarchy issues, which are critical to get performance on advanced RISC processors. Recently, [Zubair and Mukkamala 2008] has developed implementations that utilize the memory hierarchy to improve performance for Binomial pricing for European options.

In this paper, we derive lower bounds on memory traffic between different levels of the hierarchy for binomial and trinomial computation. We also give algorithms for binomial and trinomial models that exhibit a constant-factor optimal amount of memory traffic between different levels. We implemented these algorithms on an UltraSparc IIIi processor with a 4-level memory hierarchy and demonstrated that our algorithms outperform algorithms without cache blocking by a factor of up to 5. Our algorithms are fast, they run in seconds for typical applications, namely, options expiring in six months in which estimates are made once per minute (about $n = 60,000$ samples). The codes for these algorithms are available at *http://www.cs.odu.edu/~zubair/OPCodes*.

The multicore processors add another dimension to the memory hierarchy. They have varying degrees of sharing of caches by cores at different levels. Most architectures have cores with a private Level-1 cache. Depending on the architecture, a Level-2 cache is shared by two or more cores and a Level-3 cache is shared by four or more cores. The main memory is typically shared by all cores. The degree of sharing at a level varies from one multicore processor to another. These characteristics have been captured in a new model, called the Unified Memory Model, which is described in [Savage and Zubair 2009; 2008].

The rest of the paper is organized as follows. The general question of designing algorithms for efficient use of memory hierarchies is discussed in Section 2. Section 3 describes the computational requirements for the two option pricing models. Section 4 develops memory traffic bounds using the memory hierarchy model. In Section 5 we propose optimal algorithms for valuing options using the binomial and trinomial option pricing models. Section 6 discusses implementation and experimental results for the proposed algorithms. In Section 7 we examine cache-oblivious algorithms for memory management. Finally, in Section 8 we draw conclusions.

2. MEMORY HIERARCHIES

Today computers have several levels of memory hierarchy. To obtain good performance on these processors it is necessary to design algorithms that minimize I/O traffic to slower memories in the hierarchy [Hennessy and Patterson 2007; Kumar et al. 1996]. A typical processor today consists of five levels of memory. The level closest to the CPU is Level-0, which is a set of registers, typically 32 to 128; Level-1 and Level-2 are cache memories; Level-3 is the main memory followed by hard disk at Level-4. Some example processors with two level caches are the Intel Pentium III, Sun Ultrasparc IIIi, and IBM Power 3. (See Table I.) As we consider problem sizes that easily fit in main memory available on current systems, we focus on a 4-level memory hierarchy (Level-0 to Level-3). Additionally, our experiments were on a machine that uses multilevel inclusion policy for its memory hierarchy. In memory hierarchies either the multilevel inclusion or exclusion policy is enforced. In the former, a copy of the value in each location in a level-l cache is maintained in all higher level caches. These copies may be dirty, that is, not currently consistent with the value in the lowest level cache containing the original, and are updated as needed. The exclusion policy, which applies to the above rules, does not reserve space for values held in lower level caches. The results are derived for this case. However, they also hold for the inclusion policy when the memory associated with

Table I. Processors with two level caches

Processor	Level-1 Cache	Level-2 Cache
Ultrasparc IIIi	64KB Data	1024KB Shared
Intel Pentium III	16KB Data	512KB Shared
IBM Power 3	64KB Data	4096 KB Shared

a cache in the lower bounds is the difference between the capacity of a cache and that of all its subcaches.

The cache blocking technique is used to reduce memory traffic to slower memories in the hierarchy [Hennessy and Patterson 2007]. Cache blocking partitions a given computation such that the data required for a partition fits in a processor cache. For computations, where data is reused many times, this technique reduces memory traffic to slower memories in the hierarchy [Hennessy and Patterson 2007]. The cache blocking technique has been extensively applied to linear algebra applications [Dongarra et al. 1990; Anderson et al. 1999; Kågström et al. 1998; Gupta et al. 1998; Goto and van de Geijn 2008; Agarwal et al. 1994a]. Since accessing data from a slower memory is expensive, an algorithm that rarely goes to slower memory performs better. Level-0 blocking helps in reducing the number of load/store instructions by bringing the data into registers and reusing it. Blocking for Level-1 and Level-2 caches increases the reuse from the respective caches and helps in reducing the traffic to the slower level of memory. The amount of memory traffic that can be reduced between different levels of memory depends on the application, memory hierarchy architecture, and the effectiveness of the blocking algorithm.

Another factor that has an influence on the memory traffic is the storage of data. [Gustavson 2003] has shown that the inefficiency due to the storage of two-dimensional arrays both in Fortran and C programming languages can be addressed using a new generalized data structure for storing these arrays. One issue with most of these algorithms is that they need to be parameterized to be able to work on different processors with different cache sizes. For this reason various researchers have explored cache-oblivious algorithms [Frigo et al. 1999; Penner 2004]. However, portability comes at a price. [Yotov et al. 2007] have experimentally demonstrated that even highly optimized cache-oblivious programs perform significantly worse than corresponding cache aware programs for dense linear algebra applications. They point to two major reasons for this performance gap: ineffective utilization of the pipeline by cache oblivious algorithms; and the inability to effectively hide memory latency by cache oblivious algorithms.

Another reason that portability reduces performance is the difficulty in blocking for Level-0 memory (registers) by cache-oblivious algorithms. A typical cache-oblivious algorithm works by recursively partitioning the computational domain until a computation size is reached that is determined by the call overheads. Stopping the recursion of a cache oblivious algorithm without being aware of the number of registers available on the processor can lead to ineffective blocking for registers. Modern compilers are capable of unrolling and performing tiling to block for registers. However, an explicit blocking for registers is required in many cases.

3. OPTION PRICING MODELS

In this section, we describe the computational requirements for option pricing for two standard models. In particular, we describe the computation for pricing a put option contract that gives the right to its holder to sell an asset whose current price is Q at a strike price $K \leq Q$ with the expiration time T. We assume that the prevailing risk-free interest rate is r, and volatility of the asset is ν. To illustrate the computation for both models, we divide the expiration time into n intervals with each time interval $dt = T/n$. For more details on these models, please refer to [Kwok 1998; Cox et al. 1979].

3.1 Binomial Model

We use a $G^{(n)}_{biop}$ with time interval $dt = T/n$ to illustrate the computation. In $G^{(n)}_{biop}$ the level increases as we go up the tree. We identify i^{th} node at level j by (j, i), where $1 \leq j \leq n+1$ and $1 \leq i \leq n+2-j$. As part of initialization we define asset and option prices at leaf nodes ($j = 1$). Asset price q^1_i at node $(1, i)$ is given by $q^1_i = Qd^n u^{2(i-1)}$, where $u = e^{\nu\sqrt{dt}}$ and $d = u^{-1}$. Here u and d indicate the fraction by which an asset can go up or down respectively in one time interval. The initial price of the option at node $(1, i)$, c^1_i, is simply the option payoff at the node, which is given by $c^1_i = MAX(K - q^1_i, 0)$. Next we iteratively compute option prices at nodes at level $j + 1$ using prices at level j as defined below.

$$c^{j+1}_i = (p_u c^j_{i+1} + p_d c^j_i) e^{-rdt} \quad (1)$$

$$q^{j+1}_i = q^j_i * u \quad (2)$$

$$c^{j+1}_i = MAX(K - q^{j+1}_i, c^{j+1}_i) \quad (3)$$

Here, c^j_i and $q^j_i = Qd^n u^{2(i-1)+j-1}$ are the option price and asset price respectively at (j, i). Also, p_u and p_d are pseudo-probabilities given by

$$p_u = \frac{e^{rdt} - d}{u - d}$$
$$p_d = 1 - p_u$$

The final output, c^{n+1}_1 is the option price at the root node. Note that computations (2) and (3) are only required for American options. From the memory traffic perspective the difference between American and European options is that American option requires access to an additional array that stores asset prices.

The computation for a call option is similar except that the expression for payoff (3) is replaced by

$$c^{j+1}_i = MAX(q^{j+1}_i - K, c^{j+1}_i)$$

In this paper, we do not make a distinction between call and put options because from the computation and memory perspective they are identical. Though it is possible to optimize computations further for special cases, for example American call options without dividends are never exercised early [Kwok 1998] and are treated as European options.

3.2 Trinomial Model

We identify i^{th} node at level j by (j,i), where $1 \leq j \leq n+1$ and $1 \leq i \leq 2n+1-2(j-1)$. As part of initialization we define asset and option prices at leaf nodes ($j=1$). Asset price q_i^1 at node $(1,i)$ is given by $Qd^n u^{i-1}$, where $u = e^{\lambda\nu\sqrt{dt}}$, and $d = e^{-\lambda\nu\sqrt{dt}}$. Here, λ is a free parameter and a value of one reduces this model to a binomial model. The initial price of the option at node $(1,i)$ is simply the option payoff at the node, which is given by $MAX(K - q_i^1, 0)$. Next we iteratively compute option prices at nodes at level $j+1$ using prices at level j as defined below.

$$c_i^{j+1} = (p_u c_{i+2}^j + p_m c_{i+1}^j + p_d c_i^j) e^{-rdt} \quad (4)$$

$$q_i^{j+1} = q_i^j * u \quad (5)$$

$$c_i^{j+1} = MAX(K - q_i^{j+1}, c_i^{j+1}) \quad (6)$$

Here, c_i^j and q_i^j are the option price and asset price respectively at (j,i); and p_u, p_m, and p_d are pseudo-probabilities given by

$$p_u = \frac{1}{2\lambda^2} + \frac{(r - \frac{\nu^2}{2})\sqrt{dt}}{2\lambda\nu}$$

$$p_m = 1 - \frac{1}{\lambda^2}$$

$$p_d = \frac{1}{2\lambda^2} - \frac{(r - \frac{\nu^2}{2})\sqrt{dt}}{2\lambda\nu}$$

Note that computations (5) and (6) are only required for American options. The final output, c_1^{n+1} is the option price at the root node. The computation for a call option is similar except that the expression for payoff is changed as described in binomial computation for the call option.

4. BOUNDS ON MEMORY TRAFFIC

The memory hierarchy model described below was introduced to develop lower bounds on the memory traffic between adjacent levels in a memory hierarchy [Savage 1995]. (See also [Savage 1998, Chapter 11].) This model is an extension of the red-blue model introduced by [Hong and Kung 1981], a game played on directed acyclic graphs with red and blue pebbles in which red (blue) pebbles denote primary (secondary) storage locations. An I/O operation occurs when a blue pebble is placed on a vertex carrying a red pebble or vice versa. The memory hierarchy game described below extends this model to multiple levels. These models have been applied to matrix multiplication, FFT, and applications involving permutations. We use the hierarchical model to derive lower bounds on a 4-level memory hierarchy for option pricing using the binomial and trinomial models. This model captures the details of memory hierarchies that suffice to make explicit the essential dependence of algorithms on the sizes of multiple caches Other models have been proposed to capture similar aspects of this problem. See for example [Savage and Vitter 1987; Aggarwal et al. 1987; Aggarwal et al. 1987; Vitter 2006].

4.1 The Memory Hierarchy Game

We assume that the capacity of memory at level-l is σ_l, for $0 \leq l \leq 2$, which is the number of words it can hold. We further assume that the caches uses the multilevel inclusion policy, which implies that data in the Level-l cache is a subset of the data in the Level-$(l+1)$ cache.

Let $T_l(\underline{\sigma}, G)$ denote the memory traffic between levels l and $l-1$ in the hierarchy where traffic is measured by the number of words that move between the levels and $\underline{\sigma}$ is a vector denoting the amount of memory available at each level. Later we derive lower bounds on $T_l(\underline{\sigma}, G)$ for $G^{(n)}_{biop}$ and $G^{(n)}_{triop}$. The key to deriving these bounds is to compute the S-span of $G^{(n)}_{biop}$ and $G^{(n)}_{triop}$, which is defined in Section 4.2.

4.1.1 *Rules of the Memory Hierarchy Game.* The 4-level Memory Hierarchy Game (MHG) is played on directed acyclic graphs (DAGs) with σ_l pebbles at level l, $0 \leq l \leq 2$, and an unlimited number of pebbles at level 3. Placement of a level-l pebble on a vertex corresponds to moving the data associated with the vertex to the level-l memory. Computations are done on data in the first-level memory. Zero level computations can only be done at a vertex if the data needed are in zero-level memory, that is, the predecessors of the vertex carry zero-level pebbles. Data is moved to level l on a vertex only from level $l-1$ or $l+1$. This is possible only if a pebble at level $l-1$ or $l+1$ resides on the vertex. The full set of rules of the MHG is given below.

R1. (Computation Step) A zero-level pebble can be placed on any vertex all of whose immediate predecessors carry zero-level pebbles.

R2. (Pebble Deletion) Except for level-3 pebbles on output vertices, a pebble at any level can be deleted from any vertex.

R3. (Initialization) A level-3 pebble can be placed on an input vertex at any time.

R4. (Input from Level-l) For $1 \leq l \leq 3$, a level-$(l-1)$ pebble can be placed on any vertex carrying a level-l pebble.

R5. (Output to Level-l) For $1 \leq l \leq 3$, a level-l pebble can be placed on any vertex carrying a level-$(l-1)$ pebble.

The MHG has **resource vector** $\underline{\sigma} = (\sigma_0, \sigma_1, \sigma_2)$, where $\sigma_j \geq 1$ for $0 \leq j \leq 2$ is the number of storage locations at level-l. As we are assuming a multilevel inclusion policy, the total number of storage words up to and including level l is also σ_l. Zero-level pebbles can slide from a predecessor to a successor vertex, which corresponds to using a register as both the source and target of an operation.

4.2 Computational Inequalities

DEFINITION 1. *The S-span of a DAG G, $\rho(S, G)$, is the maximum number of vertices of G that can be pebbled in a zero-level pebble game starting with any initial placement of S red pebbles.*

Once we have the S-span of a dag, we apply the following results of MHG developed first in [Savage 1995] and strengthened in [Savage and Zubair 2009] to derive lower bounds on $T_l(\underline{\sigma}, G)$.

THEOREM 4.1. *Consider a pebbling of the DAG G with n input and m output vertices in an L-level memory hierarchy game under the multilevel inclusion policy. Let $\rho(S, G)$ be the S-span of G and $|V^*|$ be the number of vertices in G other than the inputs. Assume that $\rho(S, G)/S$ is a non-decreasing function of S.*

Then, for $0 \leq l \leq L-1$ the communication traffic between the lth and $(l-1)$st levels, $T_l^{(L)}(\underline{\sigma}, G)$, satisfies the following lower bound where $\sigma_{(l-1)}$ is the number of pebbles at level $l-1$.

$$T_l^{(L)}(\underline{\sigma}, G) \geq \frac{\sigma_{(l-1)}|V^*|}{\rho(2\sigma_{(l-1)}, G)}$$

It is also trivially true that $T_l^{(L)}(\underline{\sigma}, G) \geq (n+m)$.

Below we derive new upper bounds on the S-span of the graphs $G_{biop}^{(n)}$ and $G_{triop}^{(n)}$ which, with the above result, provides new lower bounds on $T_l^{(L)}(\underline{\sigma}, G_{biop}^{(n)})$ and $T_l^{(L)}(\underline{\sigma}, G_{triop}^{(n)})$. Note that $|V^*| = n(n+1)/2$ and $|V^*| = n^2$ for the two graphs, respectively.

4.3 Lower Bounds for $G_{biop}^{(n)}$

DEFINITION 2. *For $i < j$, let p_i^j denote a path in a $G_{biop}^{(n)}$ from a node at level i to a node at level j containing a sequence of nodes $x_i, x_{i+1}, \ldots x_j$. (A leaf vertex is at level 1.) The length of a path is the number of edges that it contains.*

LEMMA 4.1 [COOK 1974]. *$G_{biop}^{(n)}$ requires a maximum of $S = n+1$ pebbles to place a pebble on the output vertex. The graph can be pebbled completely with $n+1$ pebbles without repebbling any vertices.*

PROOF. Initially all vertices are unpebbled and all paths from inputs to the output are pebble-free. There is some last path p_1^{n+1} from an input to the output that is free of pebbles. This path has $n+1$ vertices. When a pebble is placed on the input to p_1^{n+1}, the graph already had pebbles on each of the paths leading to each of the n other vertices on p_1^{n+1}. Thus, when the input to p_1^{n+1} is pebbled, the graph has at least $n+1$ pebbles on it.

To show that the graph can be pebbled completely without repebbling any vertices, place all $n+1$ pebbles on the inputs, slide the leftmost pebble up one level and then proceed to slide the remaining pebbles up one level to pebble the leaves of the subgraph $G_{biop}^{(n-1)}$ with n leaves. The rest follows by induction. □

THEOREM 4.2. *The S-Span of $G_{biop}^{(n)}$ satisfies $\rho(S, G_{biop}^{(n)}) \leq S(S-1)/2$.*

PROOF. Let V be the set of vertices that are pebbled from an initial placement of S pebbles. These are vertices that are pebbled without using any of the caches. V consists of one or more disjoint sets of vertices. Let $V_1, V_2, \ldots, V_m$ be these sets and let $s_1, s_2, \ldots, s_m$ be the number of pebbles placed on the vertices of these sets initially. Then, $S = \sum_i s_i$.

Consider one of these sets, say V_i. Let k_i be the number of vertices on the longest directed path p^* in this set. (Edges are directed from leaves toward the output.) Let v_L be the last vertex on p^*. It follows that V_i cannot be larger than the subgraph $G_{biop}^{(k_i)}$ that contains p^*.

Using the argument of Lemma 4.1, there must be at least k_i pebbles in V_i when the last path to v_L is closed. Thus, $s_i \geq k_i$. Furthermore, it is possible to pebble all vertices in $G_{biop}^{(k_i)}$ with k_i pebbles. The number of vertices that are pebbled in V_i other than the vertices carrying pebbles initially is at most $k_i(k_i-1)/2$.

The total number of vertices that can be pebbled, τ, satisfies $\tau \leq \sum_i t_i \leq \sum_i k_i(k_i-1)/2$. Using the identity $a(a-1)+b(b-1) \leq (a+b)((a+b)-1)$, where $a, b \geq 0$, it follows that $\tau \leq S(S-1)/2$. □

Applying Theorem 4.1 we have the following result.

THEOREM 4.3. *The computation of $G_{biop}^{(n)}$ on a 4-level memory hierarchy system with storage σ_l at level l or less satisfies the following lower bound on memory traffic for $1 \leq l \leq 3$.*

$$T_l(\underline{\sigma}, G_{biop}^{(n)}) \geq \frac{n(n+1)}{4(\sigma_{l-1}-1))}$$

4.4 Lower Bounds for $G_{triop}^{(n)}$

In the case of trinomial option pricing, the memory traffic is governed by the trinomial graph shown in Figure 2.

LEMMA 4.2. *$G_{triop}^{(n)}$ requires $S = 2n+1$ pebbles to pebble the output vertex. It can be pebbled completely with $2n+1$ pebbles without repebbling any vertices.*

PROOF. Initially all vertices are unpebbled and all paths from inputs to the output are pebble-free. There is some last path p_1^{n+1} from an input to the output that is free of pebbles. This path has n internal vertices plus the leaf vertex. When a pebble is placed on the input to p_1^{n+1}, the graph already had pebbles on each of the paths leading to each of the n other vertices on p_1^{n+1}. Since there are two such paths per vertex, when the input to p_1^{n+1} is pebbled, the graph has at least $2n+1$ pebbles on it.

The graph can also be pebbled with $2n+1$ pebbles without repebbling vertices by sliding the pebbles up one level starting with the leftmost pebble and proceeding to the right. A total of $(n+1)^2$ steps is needed. □

THEOREM 4.4. *The S-Span of $G_{triop}^{(n)}$ satisfies $\rho(S, G_{triop}^{(n)}) \leq (S-1)^2/4$.*

PROOF. The proof of this result is similar to that of Theorem 4.2. The only difference is that the number of vertices that can be pebbled in $G_{triop}^{(k)}$ with $2k+1$ leaves from a starting position in which $2k+1$ pebbles reside on the leaves is $t = (2(k-1)+1) + (2(k-2)+1) + \cdots + (2(0)+1) = k^2$, as we now show. If the ith set, V_i, which has s_i pebbles on it initially, has a longest path of length k_i (it has k_i+1 vertices), then $s_i \geq 2k_i+1$. A total of τ vertices can be pebbled where τ satisfies $\tau \leq \sum_i t_i \leq \sum_i k_i^2$. Since $k_i \leq (s_i-1)/2$, $\tau \leq \sum_i (s_i-1)^2/4$. Using the identity $a^2+b^2 \leq (a+b)^2$, where $a, b \geq 0$, it follows that $\tau \leq (S-1)^2/4$. □

Again applying Theorem 4.1, we have the following result.

THEOREM 4.5. *The computation of $G_{triop}^{(n)}$ on a 4-level memory hierarchy system with storage σ_l at level l or less satisfies the following lower bound on memory traffic*

for $1 \leq l \leq 3$.

$$T_l(\underline{\sigma}, G^{(n)}_{triop}) \geq \frac{4n^2}{(\sigma_{l-1} - 1)}$$

5. OPTIMAL ALGORITHMS

We now develop cache blocking algorithms that greatly reduce the traffic between levels in a memory hierarchy. They are based on the recursive partition of the computation into smaller blocks, where blocks at each level of recursions fit into the corresponding level of memory hierarchy. Partitioning at the first recursion results in blocks that fit in the Level-2 cache; partitioning at the next recursion results in blocks that fit in the Level-1 cache, and so on. This partitioning ensures that once we bring the required data for a block into a faster memory in hierarchy, we reuse the data a sufficient number of times before bringing in the data for the next block. This reduces the traffic between different levels of the memory hierarchy. The algorithm for binomial option pricing is similar to the one proposed in [Zubair and Mukkamala 2008] for a single level cache for European option pricing. However, the algorithm we propose in this paper for binomial option pricing works for multiple levels of the memory hierarchy.

5.1 Binomial Option Pricing

We present a multilevel algorithm (Algorithm 1), the outermost loop of which is shown below. This algorithm assumes that data in $G^{(n)}_{biop}$ is recursively partitioned into blocks at different levels of the memory hierarchy and that the computation is done from the leftmost leaf. We could equally well have chosen to compute from the rightmost leaf.

The first level of partitioning is for the Level-2 cache and is illustrated in Figure 3. $G^{(n)}_{biop}$ is partitioned into rhombuses (or partial rhombuses, namely, triangles) $b^1_{j,i}$ of size m (there are m vertices on each side) for $1 \leq j \leq (n+1)/m$, and $1 \leq i \leq (n+1)/m - j + 1$. To keep our description simple, we treat triangles as complete rhombuses and assume that m evenly divides $n+1$. The run time T_R is increased by a fraction of approximately $m/(n+1)$. (Let $\alpha = (n+1)/m$. $G^{(n)}_{biop}$ has $\alpha(\alpha+1)/2$ blocks, all of which are rhombuses except for the α triangles on the left boundary. Replacing these triangles by rhombuses increases the number of operations by $\alpha(m^2/2)$, a fraction of $1/(\alpha+1)$ of the total.) Replacing triangles by rhombuses has no effect on the number of I/O operations.

Algorithm 1: Processes $G^{(n)}_{biop}$

```
for i = 1 to (n+1)/m do
    for j = 1 to i do
        processRhombus(b^1_{j,i}, m)
    end
end
```

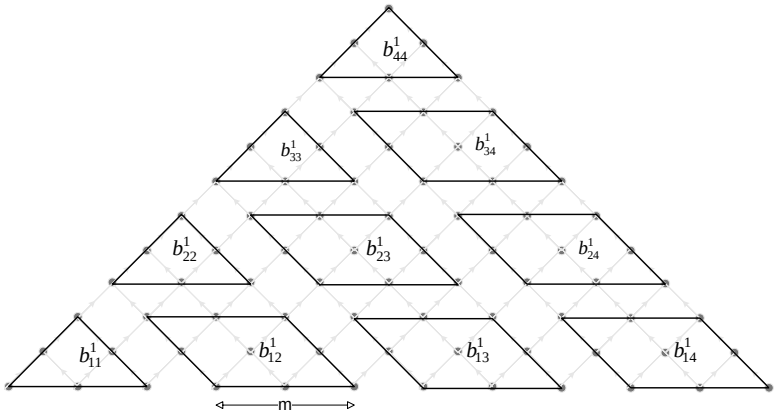


Fig. 3. $G^{(n)}_{biop}$ partitioned into blocks of size m

The algorithm processes the blocks in the order $b^1_{1,1}$, $b^1_{1,2}$, $b^1_{2,2}$, $b^1_{1,3}$, $b^1_{2,3}$, $b^1_{3,3}$, etc. That is, the blocks are processed along diagonals that slant up to the left. We could also process the blocks by rows, that is, in the order $b^1_{1,1}$, $b^1_{1,2}$, $b^1_{1,3}$, $b^1_{1,4}$, $b^1_{2,2}$, $b^1_{2,3}$, etc. Both orders create the same amount of memory traffic.

To pebble the vertices in block $b^1_{j,i}$ requires that the Level-2 cache contain the boundary values (each value consists of two quantities, c^j_i and q^j_i) on the top side of the block below it, $b^1_{j-1,i}$, the boundary values on the right side of the block to its left, $b^1_{j,i-1}$, and the upper-right corner value of the block below the latter block, $b^1_{j-1,i-1}$. Hence, the total number of values needed to process a block[1] of size m is $2m$, which is also the storage requirement for the Level-2 cache, σ_2. Algorithm 1 is a high-level description of the outermost algorithm recursion. We ignore the special handling of the first row of blocks as their first row does not require any computation.

To handle data movement between the Level-1 and Level-2 caches, each $m \times m$ rhombus is decomposed into $(m/q)^2$ $q \times q$ rhombuses. By the reasoning given above the storage requirement for Level-1 cache is $\sigma_1 = 2q$. Finally, each $q \times q$ rhombus is decomposed into $(q/r)^2$ $r \times r$ rhombuses. The storage requirement for Level-1 cache is $\sigma_0 = 2r$. Here we assume that r divides m and q. A high-level description of the second algorithm is given in Algorithm 2.

Memory Traffic. We estimate the memory traffic between main memory (Level-3 cache) and the Level-2 cache by observing that while processing a typical block, we replace m values in Level-2 cache resulting in a total I/O traffic of $2m$ values. Note that the total I/O traffic for the first block in the second nested loop of Algorithm1. is $4m$ as the number of values that need to be replaced for processing this block is $2m$. However, the memory traffic is dominated by processing of typical blocks with I/O traffic of $2m$ values per block. To get the total traffic we need to multiply the

[1]Strictly speaking the number of q^j_is needed to process a block is one less than the required number of c^j_is.

Algorithm 2: Processes processRhombus($b^1_{j_1,i_1}$, m)

```
for i = 1 to m/q do
    for j = 1 to m/q do
        processRhombus(b^2_{j,i}, q)
    end
end
```

average memory traffic for processing a block by the number of blocks of size m. It follows that the total memory traffic between Level-3 (main memory) and Level-2 cache, T_3, is given by

$$T_3 \approx 2m\left(\frac{n^2}{2m^2}\right) = \frac{n^2}{m}$$

Using the same reasoning we can get an estimate for T_2 and T_1.

$$T_2 \approx 2q\left(\frac{n^2}{2m^2}\right)\left(\frac{m^2}{q^2}\right) = \frac{n^2}{q}$$

$$T_1 \approx 2r\left(\frac{n^2}{2m^2}\right)\left(\frac{m^2}{q^2}\right)\left(\frac{q^2}{r^2}\right) = \frac{n^2}{r}$$

From Theorem 4.3 we observe that T_3, T_2, and T_1 for the proposed algorithm are optimal to within a constant factor of 8.

5.2 Trinomial Option Pricing

Without loss of generality, the trinomial option pricing algorithm is evaluated from the leftmost vertex. The blocking algorithm for $G^{(n)}_{triop}$ is similar to the blocking algorithm of $G^{(n)}_{biop}$. It recursively partitions the $G^{(n)}_{triop}$ to block for different levels of the memory hierarchy. The first level of partitioning for Level-2 cache is illustrated in Figure 4. All blocks have a rhombus shape except those along the left edge of $G^{(n)}_{triop}$. As in the case of $G^{(n)}_{biop}$, we treat an incomplete rhombus as a complete one and assume that m divides $2n+1$. Using similar reasoning as for the binomial, we find the total number of values needed to process a block of size m is $\sigma_2 = 3m$.

The order in which the rhombuses are visited makes a difference for this case although it has no effect for the binomial option pricing model. If the rhombuses are visited on diagonals slanting up and to the left, $4m$ I/O operations are required for each rhombus. However, if the rhombuses are visited by rows, only $2m$ I/O operations are required. We have implemented both methods. A high-level description of the outermost algorithm recursion for the first ordering is given in Algorithm 3. We ignore the special handling of the first row of blocks.

To handle data movement between the Level-1 and Level-2 caches, we partition a rhombus of size m into $(m/q)^2$ $q \times q$ rhombuses $b^2_{j,i}$ such that all the required data fits in Level-1 cache, that is $\sigma_1 = 3q$. A high-level description of the second recursion is given in Algorithm 4. For the final recursion, we partition each rhombus of size q into $r \times r$ rhombuses. The amount of storage needed at Level-0 cache is $\sigma_0 = 3r$.

Algorithm 3: Processes $G^{(n)}_{triop}$

```
for i = 1 to (2n + 1)/m do
    for j = 1 to ⌈i/2⌉ do
        processRhombus(b^1_{j,i}, m)
    end
end
```

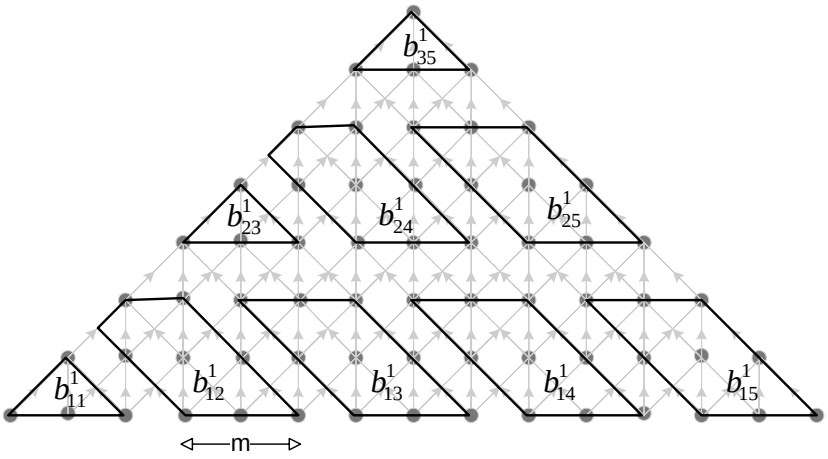


Fig. 4. $G^{(n)}_{triop}$ partitioned into blocks of size m

Algorithm 4: Processes processRhombus($b^1_{j_1,i_1}$, m)

```
for i = 1 to m/q do
    for j = 1 to m/q do
        processRhombus(b^2_{j,i}, q)
    end
end
```

Memory Traffic. We estimate the memory traffic between main memory (Level-3 cache) and the Level-2 cache for the two orderings. For the first ordering, while processing a typical block we replace $2m$ values in Level-2 cache resulting in a total I/O traffic of $4m$ values. For the second ordering the I/O traffic per block consists of $2m$ values. Note that the total I/O traffic for blocks on the left boundary is $6m$ because the number of values that need to be replaced to process them is $3m$. However, the memory traffic is dominated by processing of typical blocks with I/O traffic of $4m$ values per block. To get the total traffic we need to multiply the average memory traffic for processing a block by the number of blocks of size m. It follows that the total memory traffic between Level-3 (main memory) and Level-2

cache for the two orderings, $T_3^{(1)}$ and $T_3^{(2)}$, is given by

$$T_3^{(1)} \approx 4m\left(\frac{n^2}{m^2}\right) = \frac{4n^2}{m}$$
$$T_3^{(2)} \approx 2m\left(\frac{n^2}{m^2}\right) = \frac{2n^2}{m}$$

Using the same reasoning we get estimates for $T_2^{(i)}$ and $T_1^{(i)}$, $i \in \{1, 2\}$.

$$T_2^{(1)} \approx 4q\left(\frac{n^2}{m^2}\right)\left(\frac{m^2}{q^2}\right) = \frac{4n^2}{q}$$
$$T_2^{(2)} \approx 2q\left(\frac{n^2}{m^2}\right)\left(\frac{m^2}{q^2}\right) = \frac{2n^2}{q}$$
$$T_1^{(1)} \approx 4r\left(\frac{n^2}{m^2}\right)\left(\frac{m^2}{q^2}\right)\left(\frac{q^2}{r^2}\right) = \frac{4n^2}{r}$$
$$T_1^{(2)} \approx 2r\left(\frac{n^2}{m^2}\right)\left(\frac{m^2}{q^2}\right)\left(\frac{q^2}{r^2}\right) = \frac{2n^2}{r}$$

From Theorem 4.5 we observe that $T_3^{(1)}$, $T_2^{(1)}$, and $T_1^{(1)}$ for the proposed algorithm are optimal to within a factor of 3 whereas $T_3^{(2)}$, $T_2^{(2)}$, and $T_1^{(2)}$ are optimal to within a factor of 1.5. This difference is due to the observation made earlier that the horizontal ordering results in memory traffic that is a factor of two less compared to the diagonal ordering.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We implemented the proposed algorithms on a Sun Workstation with Solaris 10 OS and UltraSPARC IIIi processor operating at 1050 MHz. The UltraSPARC IIIi processor has a 64 KB Level-1 data cache organized as 4-way set associative and has a line size of 32 bytes. The data cache can hold up to 8K double precision floating-point data. The processor also has a Level-2 cache of 8 MB which is shared between instruction and data. As we consider problem sizes up to a maximum of 64K leaf nodes for both models, we can accommodate all the required data in Level-2 cache. Thus for our experimentation, we ignored partitioning for Level-2.

The UltraSparc IIIi processor executes two floating-point instructions in one cycle, so the peak performance of the processor is 2.1 GFLOPS. To evaluate the performance of various algorithms, we use wall clock execution time. To evaluate how well a given algorithm matches the underlying architecture, we also compute algorithm performance as the percentage of the theoretical peak performance for the target machine. For example, when we get 1.05 GFLOPS on the Sun Workstation, our code is running at 50% of the peak. All our algorithms were implemented using Fortran 90/95. We compiled all our code including vanilla code with the "-fast" option, which combines various complementary optimizations for the target processor [Garg and Sharapov 2001]. Where specified, for some results we also used the "-nodepend" option to turn off compiler based cache blocking. As mentioned earlier, to keep our implementation simple we treat incomplete rhombuses as complete rhombuses. We handle this in the implementation by padding the arrays,

Table II. Performance of the optimal algorithm for binomial option pricing when rhombuses are visited on diagonals slanting up and to the left

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	0.14	1490	71.0%
16384	0.55	1501	71.5%
32768	2.17	1508	71.8%
65536	8.60	1511	71.9%

Table III. Performance of the optimal algorithm for binomial option pricing when rhombuses are visited by rows

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	0.14	1496	71.2%
16384	0.55	1506	71.7%
32768	2.16	1512	72.0%
65536	8.57	1515	72.1%

storing asset and option price with zeros. This does not have a significant impact on the performance numbers reported in this paper.

The performance results for the binomial optimal algorithm when rhombuses are visited on diagonals slanting up and to the left are summarized in Table II. We also include the results for the binomial optimal algorithm when rhombuses are visited by rows in Table III. As expected, the results are similar for both the algorithms. For these results, we compiled the code using the "-fast" and "nodepend" options of the Sun compiler. The first column indicates problem size (we use the number of leaves in the binomial model as the problem size). Observe that we obtained around 70% of the peak performance for this algorithm. One major reason we are not doing better than 70% of the peak is due to the the use of the MAX function inside the nested loop (equation (3) in Section 3). The use of this function creates a bottleneck for the pipeline as it results in a branch instruction inside the nested loop [Hennessy and Patterson 2007].

The results reported in the Table II are for a block size of 512 for Level-1 cache blocking. This value was chosen as a result of an experiment. The maximum size of Level-1 block is determined by the size of data cache on the processor. Figure 5 plots execution time for various block sizes for $n = 65536$. From the figure it is clear that once we go beyond a block size of 4096, the performance drops. We selected the block size of 512, which is in the acceptable range.

For Level-0 blocking, we selected a block of size 4. The maximum size of Level-0 blocking is determined by the number of registers available on the processor. A register tiling improves the ratio of the number of floating point operations to the number of loads/stores. To understand this, consider Algorithm 5 for a 2×2 register

tiling based on our optimal algorithm. Note that lines 3 to 9 corresponds to load data from Level-1 cache into registers; and lines 22 to 27 correspond to stores from register into cache. The rest of the lines 10-21 correspond to computations. Observe that a 2×2 tiling improves the ratio of the number of floating point operations to the number of loads/stores from 6/5 to 24/13 (a MAX operation is counted as two floating point operation); and it also increases the number of instructions inside the main body giving flexibility in scheduling to minimize data dependency. To see the number of floating point and load/stores operations for an algorithm without tiling, please refer to equations (1)-(3). We have three floating point operations due to (1). Note we do not count multiplication with an exponential as it is done outside the inner loop. There is one floating point operation for (2), and two for (3) resulting in a total of 6 floating point operations. There are three loads for c_i^j, c_{i+1}^j, and q_i^j; and two stores for c_i^{j+1}, and q_i^{j+1} for a total of 5 loads/stores. The count for 2×2 tiling can be seen from Algorithm 5. A 4×4 tiling increases the ratio further to 96/29. In general, for a $r \times r$ tiling this ratio is given by $6r^2/(8r-3)$. However, as we increase tiling we increase the required register count. Once we increase tiling beyond the number of available registers on the processor, we start observing spilling in the compiled code that offsets the advantage of tiling. Spilling occurs when the compiler transfers some variables from registers to cache, resulting in slower access. In other words, the number of available registers in the processor limits the largest amount of tiling. For our implementation, we found 4×4 tiling to be optimal. It should be mentioned here that modern compilers are capable of loop unrolling and in general are efficient. However, for nested loops with varying bounds like in our case, a user with a knowledge of runtime constraints can do a better job of unrolling.

For comparison, we implemented a vanilla algorithm, which refers to a straightforward implementation of binomial option pricing without any explicit cache blocking for Level-0 and Level-1. The performance results for the vanilla algorithm are summarized in Table IV. For these results, we compile the vanilla code with "-fast" and "-nodepend" options of the Sun compiler. The first column indicates problem size (note that we use the number of leaves in the binomial model as the problem size). Observe that the vanilla algorithm's performance varies from 8% to 9% of the peak performance as compared to the proposed optimal algorithm that achieves approximately 70% of the peak performance. We also compiled the vanilla code with the "-fast" option and without the "-nodepend" option, thus letting the Sun compiler do cache blocking and unrolling. The results of this experiment are summarized in Table V. Observe, that the compiler-based cached blocking does improve the performance but is still a factor of 2 lower than the optimal algorithm.

The performance results for the two variations of the optimal algorithm for the trinomial model are summarized in Tables VI and VII. Observe that there is a slight improvement for the case when rhombuses are visited by rows, but not significant. We suspect that this is due to the interplay between various parameters such as set-associativity, block size, and replacement policy. For comparison we include the results for the vanilla algorithm in Tables VIII and IX. The optimal algorithm performance results shown in Tables VI and VII are for a block size of 255 for Level-1 blocking. As in the binomial implementation, we treat all blocks as complete

Algorithm 5: A 2×2 Register Tiling

for $(j \leftarrow 1; j \leq m; j \leftarrow j+2)$ **do**
  for $(i \leftarrow 1; i \leq m; i \leftarrow i+2)$ **do**
    $x_1 \leftarrow c_{i-1}^{j}$
    $x_2 \leftarrow c_{i}^{j-1}$
    $x_3 \leftarrow c_{i+1}^{j-1}$
    $x_4 \leftarrow c_{i+2}^{j-1}$
    $y_1 \leftarrow q_{i-1}^{j}$
    $y_2 \leftarrow q_{i}^{j-1}$
    $y_3 \leftarrow q_{i+1}^{j-1}$
    $x_2 \leftarrow p'_u x_3 + p'_d x_2$
    $x_3 \leftarrow p'_u x_4 + p'_d x_3$
    $y_2 \leftarrow y_2 * u$
    $y_3 \leftarrow y_3 * u$
    $x_2 \leftarrow MAX(K - y_2, x_2)$
    $x_3 \leftarrow MAX(K - y_3, x_3)$
    $x_1 \leftarrow p'_u x_2 + p'_d x_1$
    $x_2 \leftarrow p'_u x_3 + p'_d x_2$
    $y_1 \leftarrow y_1 * u$
    $y_2 \leftarrow y_2 * u$
    $x_1 \leftarrow MAX(K - y_1, x_1)$
    $x_2 \leftarrow MAX(K - y_2, x_2)$
    $c_{i-1}^{j+1} \leftarrow x_1$
    $c_{i}^{j+1} \leftarrow x_2$
    $c_{i+1}^{j} \leftarrow x_3$
    $q_{i-1}^{j+1} \leftarrow y_1$
    $q_{i}^{j} \leftarrow y_2$
    $q_{i+1}^{j} \leftarrow y_3$
  end
end

Table IV. Performance of the vanilla algorithm for binomial option pricing with "-fast -nodepend" compiler options (compiler based cache blocking turned off)

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	0.45	506	24.1%
16384	2.09	409	19.5%
32768	8.86	375	17.9%
65536	35.75	366	17.4%

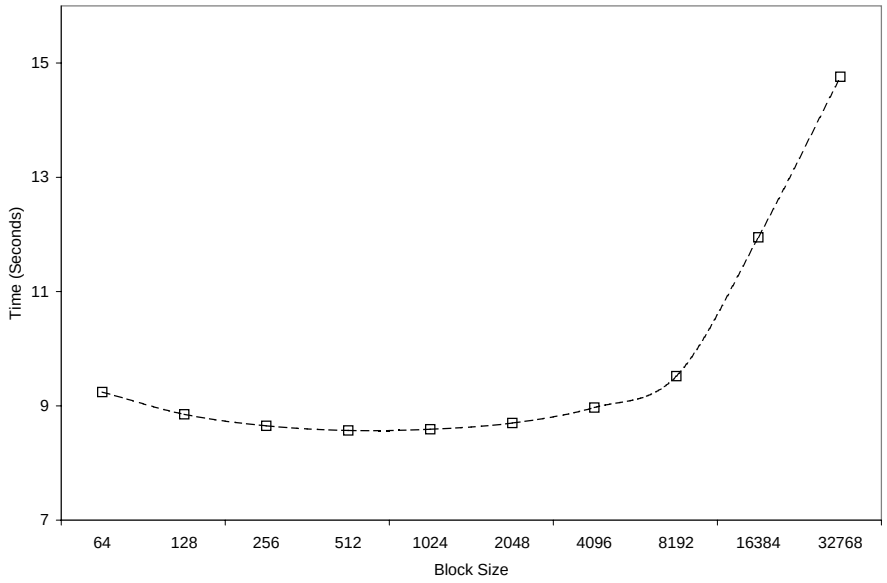


Fig. 5. Performance of optimal binomial pricing algorithm for various blocks sizes for $n = 65536$

Table V. Performance of the vanilla algorithm for binomial option pricing with "-fast" compiler option (compiler based cache blocking turned on)

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	0.32	719	34.2%
16384	1.19	718	34.2%
32768	4.62	719	34.2%
65536	18.20	719	34.2%

Table VI. Performance of the optimal algorithm for trinomial option pricing when rhombuses are visited on diagonals slanting up and to the left

2n+1	Execution Time (sec)	MFLOPS	% Peak
8415	0.11	1450	69.0%
16575	0.40	1453	69.2%
31875	1.44	1455	69.3%
64515	5.80	1457	69.4%

rhombuses by padding the arrays with zeroes. For Level-0 blocking we use a block size of 5 resulting in a 5×5 register tiling.

Table VII. Performance of the optimal algorithm for trinomial option pricing when rhombuses are visited by rows

2n+1	Execution Time (sec)	MFLOPS	% Peak
8415	0.10	1514	72.1%
16575	0.39	1498	71.4%
31875	1.41	1486	70.8%
64515	5.72	1479	70.4%

Table VIII. Performance of the vanilla algorithm for trinomial option pricing with "-fast -nodepend" compiler options (compiler based cache blocking turned off)

2n+1	Execution Time (sec)	MFLOPS	% Peak
8415	0.26	650	31.0%
16575	1.22	493	23.5%
31875	4.87	438	20.8%
64515	20.28	420	20.0%

Table IX. Performance of the vanilla algorithm for trinomial option pricing with "-fast" compiler option (compiler based cache blocking turned on)

2n+1	Execution Time (sec)	MFLOPS	% Peak
8415	0.17	902	42.9%
16575	0.61	944	45.0%
31875	2.17	966	46.0%
64515	8.65	978	46.6%

7. CACHE-OBLIVIOUS IMPLEMENTATION

A cache-oblivious algorithm [Frigo et al. 1999] is one that does not depend on cache parameters such as cache size. Observe that our proposed algorithms partition the computation into blocks of sizes that fit in cache. In other words, we need to know the cache size to implement our optimal algorithms. The number of levels in the memory hierarchy determines the number of recursions in our algorithms. We show that our algorithms can be made independent of these parameters. We also prove that the adapted algorithms are optimal for the cache-oblivious model. The cache-oblivious model consists of a processor with one level ideal cache of size Z (it holds Z words) and a line size of L. The performance of a cache-oblivious algorithm is measured by the number of cache misses it experiences. When a cache miss occurs, a line of L words is retrieved from secondary memory. Each cache miss requires

that data be retrieved from secondary memory. Note also that to create space in the cache, a like amount of data generally has to be moved to secondary storage, a fact that may double the amount of I/O. For more details, see [Frigo et al. 1999].

A Cache-Oblivious Algorithm for the Binomial Model. For simplicity we describe the algorithm for a complete rhombus that contains the binomial pyramid of N leaves[2], where N is a power of 2. The algorithm, similar to the cache-aware algorithms proposed in this paper, recursively partitions the computation into smaller blocks. Here we partition a rhombus into four rhombuses of equal sizes. That is, a rhombus of size N at the start of recursion is partitioned into four rhombuses, $R^1_{i,j}$ of size $N/2$ each, for $1 \leq i, j \leq 2$. We continue the recursion until we reach a small size determined by the call overheads. We illustrate the recursive kernel in Algorithm 6, where we stop the recursion when we have a rhombus of size 1. Note that the first call to the recursive algorithm is with $l = 0$, and the last call with $l = \log_2 N - 1$.

Algorithm 6: Recursive processRhombus($R^l_{i,j}$, $\frac{N}{2^l}$)

```
1  l ← l + 1
2  if N/2^l < 1 then
3      for i = 1 to 2 do
4          for j = 1 to 2 do
5              processRhombus(R^l_{i,j}, N/2^l)
6          end
7      end
8  else
9      Compute option price for single node using binomial computation
       (Equations 1-3)
10 end
```

Analysis. Let $P(N)$ and $M(N)$ be estimates for the number of cache misses for a pyramid of size N and the containing rhombus. We derive a bound on $M(N)$ and estimate $P(N)$ by $M(N)/2$ because the rhombus contains about twice as many vertices as the pyramid. We now obtain an approximate bound to $M(N)$ when the cache has size Z. Clearly, $M(N) \leq 4M(N/2)$. At some point in the recursion, l, we reach a stage where we partition the rhombus of size $Z/2$ into four rhombuses $R^1_{i,j}$ of size $Z/4$ each, for $1 \leq i, j \leq 2$. Observe that data required to process a rhombus of size $Z/4$ fits in the cache of size Z.

We observe that the four rhombuses can be processed in the order $R^l_{1,1}$, $R^l_{1,2}$, $R^l_{2,1}$, $R^l_{2,2}$ or the order $R^l_{1,1}$, $R^l_{2,1}$, $R^l_{1,2}$, $R^l_{2,2}$. Note that in the algorithm outlined above we use the first ordering. In both cases, Z load operations are needed on the first and third rhombuses and $Z/2$ I/O operations on the other two. This is an average of $3Z/4$ I/O operations. Thus, we estimate $M(Z/4)$ by $3Z/(4L)$ because

[2]The graph $G^{(n)}_{biop}$ has $N = n + 1$ leaves where n is the depth of the graph.

the $3Z/4$ I/O operations require a minimum of $3Z/(4L)$ cache misses, which is achieved when each line that is retrieved contains relevant data. The recurrence for $M(N)$ follows.

$$M(N) \leq \begin{cases} 3Z/(4L) & \text{if } N = Z/4 \\ 4M(N/2) & \text{otherwise} \end{cases}$$

When N is a power of two, $M(N) \leq 4^k M(N/2^k)$ where $N/2^k = Z/4$ or $2^k = 4N/Z$. Using $M(Z/4) = 3Z/(4L)$, we have $M(N) \leq 12N^2/(ZL)$. The estimate for $P(N)$ becomes $P(N) \leq 6N^2/(ZL)$.

To see how close this is to optimal, we compute the lower bound on the number of I/O operations using the results of Theorem 4.3. Since this bound takes into account both inputs to and outputs from the cache in terms of number of values (where a value consists of two words), we obtain a lower bound of $N^2/(4Z)$ on the number of input operations in number of words. This translates into $N^2/(4ZL)$ misses when each miss results in retrieving L relevant quantities, an assumption we make. Thus, the proposed cache-oblivious algorithm is optimal within a factor of 24. Observe that the cache-aware algorithm proposed in Section 5.1 incurs $2N^2/ZL$ misses on the cache-oblivious model, which is a factor of 3 improvement over the cache-oblivious algorithm. Below we show that this bound can improved by subdividing each rhombus into more rhombuses.

For this reason and others as explained shortly, we believe the performance of the cache-oblivious algorithm will not be as good as that of the algorithm implemented in this paper. The other major reason is the lack of appropriate register tiling in cache-oblivious algorithms as this would require the awareness of Level-0 size (number of registers). For example, if we stop the recursion for the cache-oblivious algorithm at some stage to enable register tiling by unrolling the code, we may perform register tiling that cannot be supported by the number of registers available in the processor. As a result there will be a number of spills resulting in performance degradation. On the other hand, if the recursion stops at a stage resulting in a register tiling of smaller size compared to what can be supported by the number of registers, we still have poor performance. In our experiments, we have observed that register tiling of the right size (that is blocking for Level-0 memory) is critical to the overall performance.

Experimental Results. We implemented the cache oblivious algorithm for Binomial computing on the Sun workstation. The performance results are summarized in Table X. Note that these results correspond to a *terminal rhombus* (rhombus when the recursion stops) of size 1. We compiled the code using "-fast" option of the Sun compiler. For comparison, we also implemented the cache aware version of our algorithm, discussed in Section 5.1, for the complete rhombus. These results are summarized in Table XI. Even with full compiler optimization the performance of the cache oblivious algorithm is quite low. One major reason is the call overheads of the recursion. We can reduce this by stopping the recursion earlier, that is when terminal rhombus is of size greater than one.

We conducted an experiment for a problem size of 65536, where we measure the performance of the cache oblivious algorithm for different terminal rhombus sizes. We plot these results in Figure 6. Note that for these results the code was compiled

Table X. Performance of the cache oblivious algorithm with terminal rhombus of size 1 for binomial option pricing

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	4.20	96	4.6%
16384	18.39	88	4.2%
32768	67.45	96	4.5%
65536	282.25	91	4.3%

Table XI. Performance of the cache aware algorithm for binomial option pricing

n+1	Execution Time (sec)	MFLOPS	% Peak
8192	0.27	1484	70.6%
16384	1.08	1489	70.9%
32768	4.32	1492	71.1%
65536	17.25	1494	71.2%

using the "-fast" option. These results indicate that the performance improves with increasing size of terminal rhombus.

We observe a maximum performance of 38% of the theoretical peak for terminal size of 128. Not all of this gain in the performance is due to the reduction of call overheads of the recursion. As the terminal size increases, the compiler is able to optimize the terminal rhombus computation by performing deep unrolling of loops and register tiling. This is obvious when we compare Figure 7 and Figure 8 corresponding to code compiled with "-fast -nodepend" and with no optimization options respectively. As mentioned earlier, "-nodepend" turns off compiler based blocking for registers and caches. Looking at these two figures, one can conclude that the initial performance gain in Figure 6 up to terminal size of 8 is mainly due to the reduction in call overheads of the recursion. The performance gain beyond terminal rhombus of size 8 is due to the compiler optimization of the terminal rhombus computation. In summary, the question what is a good terminal rhombus size for the recursion to stop cannot be fully answered by ignoring the underlying processor architecture. This can limit the performance gain of a true cache oblivious algorithm.

A Cache-Oblivious Algorithm for the Trinomial Model. We cannot use the approach outlined for binomial computation to derive the cache oblivious algorithm for trinomial computation. Observe that the number of leaves in a trinomial pyramid is not a power of 2 and is given by $N = 2n + 1$; and there is no complete rhombus as in the case of binomial computation that contains the trinomial pyra-

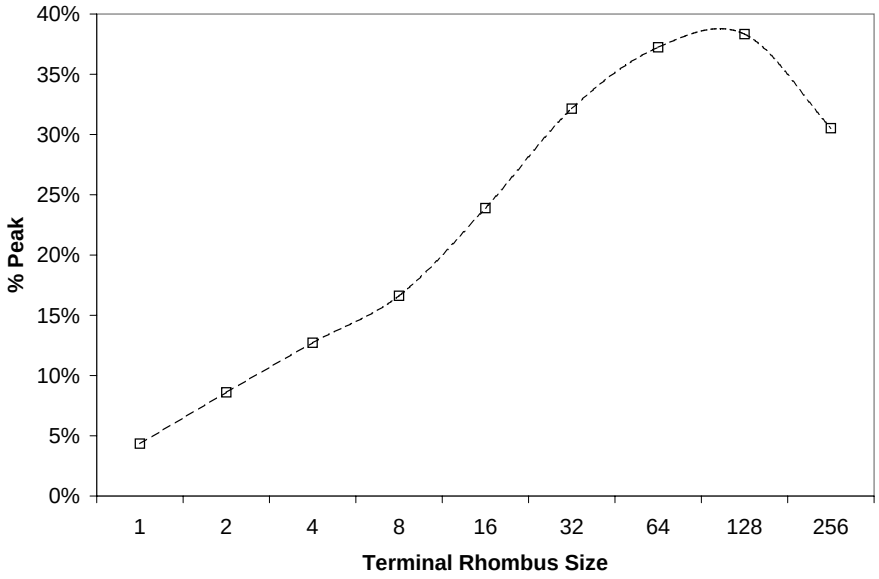


Fig. 6. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using "-fast" compiler option

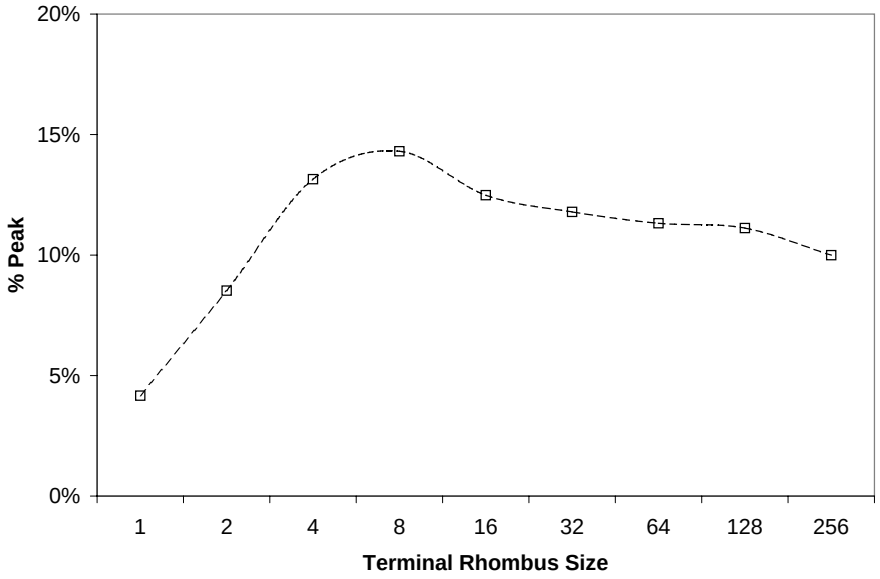


Fig. 7. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using "-fast -nodepend" compiler option

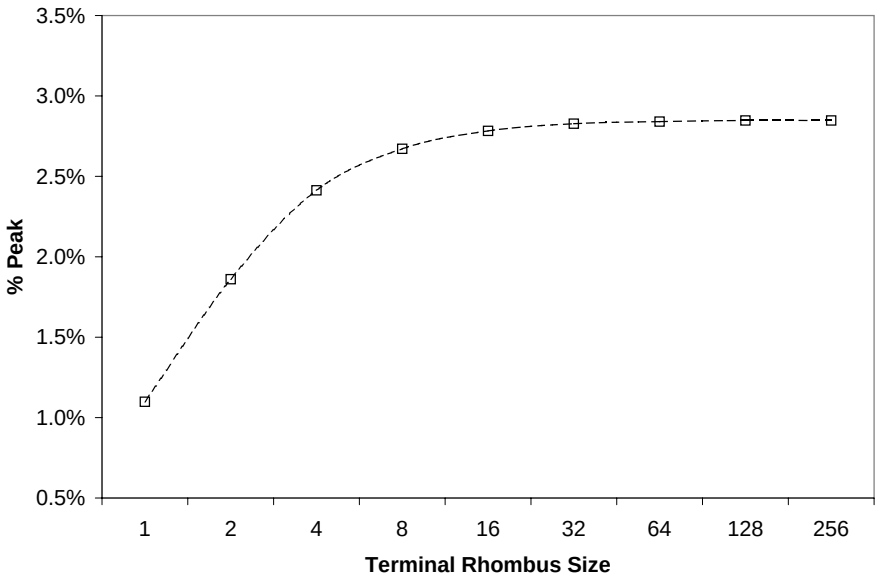


Fig. 8. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using no compiler optimization option

mid of N leaves. However, we can partition the parallelogram of size, $n \times 2n + 1$, containing the trinomial pyramid of N leaves into two rhombuses of size $n \times n$ each with one edge of nodes shared by both rhombuses. For the sake of keeping our implementation simple, we can restrict n to be a power of 2; and thus transform the trinomial computation into a processing of two rhombuses of sizes that are a power of two. We can recursively partition these rhombuses as in the case of binomial computation to derive a cache oblivious algorithm for trinomial computation. We now outline the recursion for trinomial computation for one of these rhombuses. We start the recursion by partitioning the rhombus of size n into four rhombuses, $R^1_{i,j}$ of size $n/2$ each, for $1 \leq i, j \leq 2$. We continue the recursion until we reach a rhombus of size 1. Note that the first call to the recursive algorithm is with $l = 0$, and the last call with $l = \log_2 n - 1$.

Analysis. For the trinomial case let $T(N)$ be the number of cache misses on a trinomial pyramid with N leaves. Observe that $T(N)$ is also the estimate for the number of cache misses in processing one of the two initial rhombuses of size $n = (N - 1)/2$.

We now obtain an approximate bound to $T(N)$ when the cache has size Z. Clearly, $T(N) \leq 4T(N/2)$. At some point in the recursion, l, we reach a stage where we partition the rhombus of size $Z/3$ into four rhombuses $R^1_{i,j}$ of size $Z/6$ each, for $1 \leq i, j \leq 2$. Observe that data required to process a rhombus of size $Z/6$ fits in the cache of size Z.

We observe that the four rhombuses can be processed in the order $R^l_{1,1}$, $R^l_{1,2}$,

Algorithm 7: Recursive processRhombus($R^l_{i,j}$, $\frac{n}{2^l}$)

```
1  l ← l + 1
2  if n/2^l < 1 then
3      for i = 1 to 2 do
4          for j = 1 to 2 do
5              processRhombus(R^l_{i,j}, n/2^l)
6          end
7      end
8  else
9      Compute option price for single node using trinomial computation
       (Equations 4-6)
10 end
```

$R^l_{2,1}$, $R^l_{2,2}$ or the order $R^l_{1,1}$, $R^l_{2,1}$, $R^l_{1,2}$, $R^l_{2,2}$. Note that in the algorithm outlined above we use the first ordering. For this ordering, $3Z/2$ load operations are needed on the first and third rhombuses and $Z/2$ load operations on the other two. This is an average of Z load operations. Thus, we estimate $T(Z/6)$ by Z/L because the Z load operations require a minimum of Z/L cache misses, which is achieved when each line that is retrieved contains relevant data. The recurrence for $T(N)$ follows.

$$T(N) \leq \begin{cases} Z/L & \text{if } N = Z/6 \\ 4T(N/2) & \text{otherwise} \end{cases}$$

The solution to this recurrence is $T(N) \leq 36N^2/(ZL)$, which is the number of cache misses for trinomial computation.

To see how close this is to optimal, we compute the lower bound on the number of I/O operations using the results of Theorem 4.5. Since this bound takes into account both inputs to and outputs from the cache in terms of number of values (where a value consists of two words), we obtain a lower bound of $N^2/(Z)$ on the number of input operations in number of words. This translates into $N^2/(ZL)$ misses when each miss results in retrieving L relevant quantities, an assumption we make. Thus, the proposed cache-oblivious algorithm is optimal within a factor of 36. Observe that cache-aware algorithm proposed in Section 5.2 incurs $1.5N^2/ZL$ misses (horizontal ordering) on the cache-oblivious model, which is a factor of 24 improvement over the cache-oblivious algorithm.

Experimental Results. We implemented the cache oblivious algorithm for Trinomial computing on the Sun workstation. The performance results are summarized in Table XII. Note that these results correspond to a terminal rhombus of size 1. We compiled the code using the "-fast" option of the Sun compiler. For comparison, we also implemented the cache aware version of our algorithm for trinomial computing, discussed in Section 5.2, for the complete parallelogram. These results are summarized in Table XIII. Similar to binomial computing, cache oblivious algorithm for trinomial computing performance is quite low. We also conducted experiments to see the impact of terminal rhombus size and the compiler optimiza-

Table XII. Performance of the cache oblivious algorithm with terminal rhombus of size 1 for trinomial option pricing

2n+1	Execution Time (sec)	MFLOPS	% Peak
8191	2.92	92	4.4%
16383	11.67	92	4.4%
32767	46.76	92	4.4%
65535	185.29	93	4.4%

Table XIII. Performance of the cache aware algorithm for trinomial option pricing

2n+1	Execution Time (sec)	MFLOPS	% Peak
8191	0.20	1371	65.3%
16383	0.78	1371	65.3%
32767	3.13	1372	65.4%
65535	12.51	1373	65.4%

tion on the performance of cache oblivious algorithm, see Figure 9 to Figure 11. These results indicate that the performance improves with increasing size of the terminal rhombus. We observe the maximum performance of 45% of the theoretical peak for terminal size of 128. As explained in the case of binomial computing, not all of this gain in the performance is due to the reduction of call overheads of the recursion. The performance gain beyond terminal rhombus of size 8 is due to compiler optimization of terminal rhombus computation.

Improving Cache-Oblivious Algorithms. It is possible to improve the miss complexity of cache oblivious algorithms and make it closer to the one of cache-aware algorithms. We explain this for the binomial model. We subdivide a rhombus into $16 = 4 \times 4$ rhombuses of equal size. The reason this helps to decrease the average number of misses is that the fraction of the rhombuses that need half of the maximum I/Os increases. As we increase the number of rhombuses into which a rhombus is subdivided, the fraction of the rhombuses that need half of the maximum I/Os approaches 1. In this case, the cache-aware and cache-oblivious algorithms can give the same performance. However, when the rhombuses are small, the control over the order with which vertices are pebbled falls into the hands of the compiler. If the compiler is not good at data ordering, the overall performance of the algorithm will suffer, as our experimental results show.

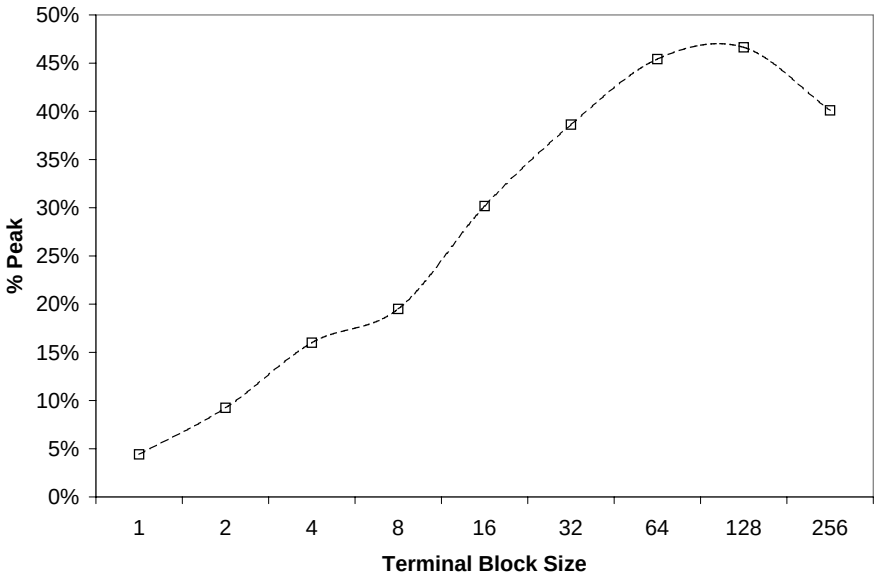


Fig. 9. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using "-fast" compiler option

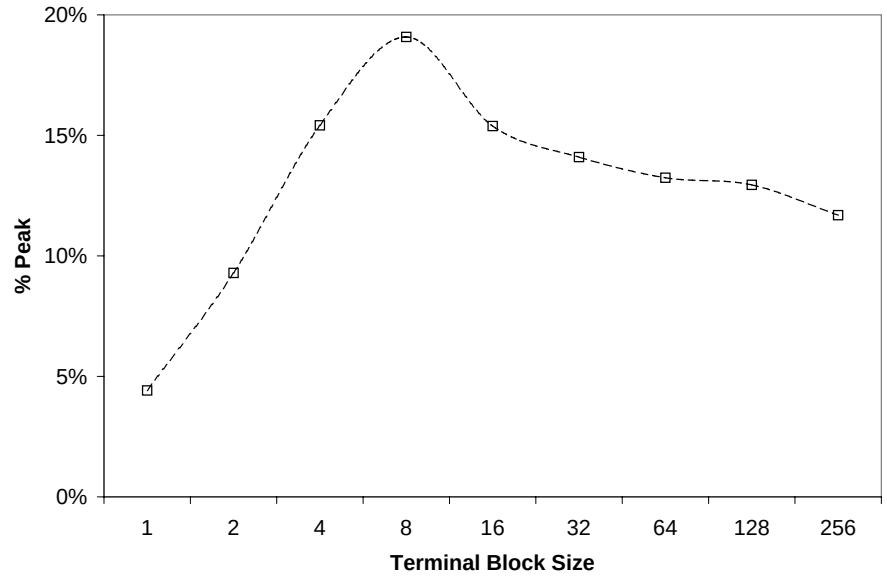


Fig. 10. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using "-fast -nodepend" compiler option

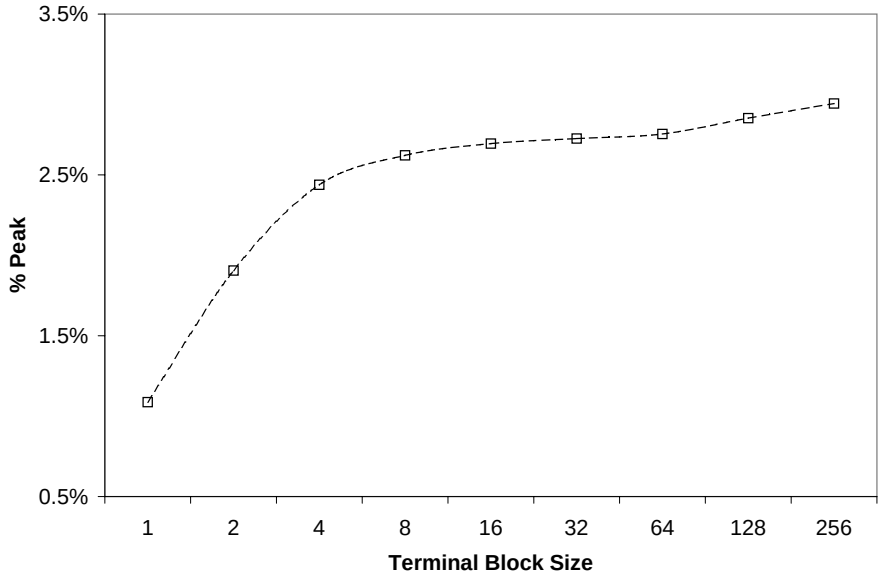


Fig. 11. Performance of the cache oblivious algorithm for various terminal rhombus sizes for $n = 65536$. These results correspond to code compiled using no compiler optimization option

8. CONCLUSIONS

We have studied the cache-efficient implementation of an important computational problem, namely, options pricing using binomial and trinomial algorithms. We have modeled them as a pebbling problem on two graphs, the binomial and trinomial pyramid graphs. We have exploited an existing framework for the study of memory hierarchies to derive lower bounds on the amount of memory traffic that is needed to pebble these graphs. This has required deriving new bounds on the S-span of these graphs.

We have also given cache-aware memory blocking algorithms to implement the option pricing computation for general memory hierarchies in which cache sizes vary by level. These blocking algorithms give memory traffic which is within a constant factor of optimal. When they are specialized to the four cache levels of the UltraSparc IIIi processor, we show that the performance improved by a factor of up to 5 and the code operated at about 70% of the peak performance. It is possible to improve the performance further by algorithmic prefetching [Agarwal et al. 1994b] and avoiding zeroes computations [Higham 2002]. We also suspect that it is possible to partition the nodes of binomial and trinomial trees, where $K - q_i^{j+1} \geq c_i^{j+1}$ in one partition, and $K - q_i^{j+1} < c_i^{j+1}$ in the other partition. This can help us in avoiding the use of MAX function inside the kernel.

We have also proposed cache-oblivious versions of our partitioning algorithm and implemented them on the Sun workstation. The performance of cache oblivious algorithm is a factor of 16 lower compared to the cache aware algorithm. We demonstrated that it is possible to improve the performance of the cache oblivi-

ous algorithm significantly if we relax the constraint on the algorithm being truly oblivious. In particular, we showed that if we determine the terminal size by experimenting on the target hardware, it is possible to get a performance of a factor of eight. However, even with this improvement the cache oblivious algorithm is a factor of two away from the cache aware algorithm.

This exercise in algorithm-specific memory blocking has applications to a broader class of options pricing models.

REFERENCES

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994a. Exploiting functional parallelism of power2 to design high-performance numerical algorithms. *IBM J. Res. Dev. 38,* 5, 563–576.

Agarwal, R. C., Gustavson, F. G., and Zubair, M. 1994b. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM J. Res. Dev. 38,* 3, 265–275.

Aggarwal, A., Alpern, B., Chandra, A., and Snir, M. 1987. A model for hierarchical memory. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 305–314.

Aggarwal, A., Chandra, A., and Snir, M. 1987. Hierarchical memory with block transfer. In *Procs. 28th Annual IEEE Symp. Foundations Computer Science*. 204–216.

Anderson, E., Bai, Z., Bischof, C. H., Blackford, S., Demmel, J., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. 1999. *LAPACK Users' Guide*, 3rd ed. SIAM, Philadelphia, PA, USA. `http://www.netlib.org/lapack/lug/`.

Cook, S. A. 1974. An observation on storage-time trade off. *J. Comp. Systems Sci 9*, 308–316.

Cox, J. C., Ross, S. A., and Rubinstein, M. 1979. Option pricing: A simplified approach. *Journal of Financial Economics 7,* 3 (September), 229–263.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software 16*, 1–28. (Algorithm 679).

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 285.

Garg, R. P. and Sharapov, I. 2001. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall PTR, New Jersey.

Gerbessiotis, A. V. 2004. Architecture independent parallel binomial tree option price valuations. *Parallel Comput. 30,* 2, 301–316.

Goto, K. and van de Geijn, R. A. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34,* 3, 1–25.

Gupta, A., Gustavson, F. G., Joshi, M., and Toledo, S. 1998. The design, implementation, and evaluation of a symmetric banded linear solver for distributed-memory parallel computers. *ACM Trans. Math. Softw. 24,* 1, 74–101.

Gustavson, F. G. 2003. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev. 47,* 1, 31–55.

Hennessy, J. L. and Patterson, D. A. 2007. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA.

Higham, D. J. 2002. Nine ways to implement the binomial method for option valuation in matlab. *SIAM Rev. 44,* 4, 661–677.

Hong, J.-W. and Kung, H. T. 1981. I/O complexity: The red-blue pebble game. In *Proc. 13th Ann. ACM Symp. on Theory of Computing*. 326–333.

Kågström, B., Ling, P., and van Loan, C. 1998. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw. 24,* 3, 268–302.

Kumar, V., Sameh, A., Grama, A., and Karypis, G. 1996. Architecture, algorithms and applications for future generation supercomputers. In *FRONTIERS '96: Proceedings of the*

6th Symposium on the Frontiers of Massively Parallel Computation. IEEE Computer Society, Washington, DC, USA, 346.

Kwok, Y. 1998. *Mathematical Models of Financial Derivatives*. Springer-Verlag, Singapore.

Penner, M. 2004. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst. 15,* 9, 769–782. Student Member-Joon-Sang Park and Fellow-Viktor K. Prasanna.

Savage, J. E. 1995. Extending the Hong-Kung model to memory hierarchies. In *Computing and Combinatorics*, D.-Z. Du and M. Li, Eds. Springer-Verlag, Lecture Notes in Computer Science, 270–281.

Savage, J. E. 1998. *Models of Computation: Exploring the Power of Computing*. Addison Wesley, Reading, Massachusetts.

Savage, J. E. and Vitter, J. S. 1987. Parallelism in space-time trade-offs. In *Advances in Computing Research*, F. Preparata, Ed. Number 4. JAI Press Inc., Greenwich, CT, 117–146.

Savage, J. E. and Zubair, M. 2008. A unified model for multicore architectures. In *Procs. 1st ACM/IEEE Int. Forum on Next-Generation Multicore/Manycore Techns.*

Savage, J. E. and Zubair, M. 2009. Evaluating multicore algorithms on a unified memory model. *Scientific Programming Journal*. To appear.

Thulasiram, R. K. and Bondarenko, D. A. 2002. Performance evaluation of parallel algorithms for pricing multidimensional financial derivatives. *icppw 00*, 306–313.

Vitter, J. S. 2006. Algorithms and data structures for external memory. *Foundation and Trends© in Theoretical Computer Science 2,* 4, 305–474.

Yotov, K., Roeder, T., Pingali, K., Gunnels, J., and Gustavson, F. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, New York, NY, USA, 93–104.

Zubair, M. and Mukkamala, R. 2008. High performance implementation of binomial option pricing. In *To Appear in Procs. of Int. Conf. on Computational Science (ICCSA 2008)*, O. Gervasi, B. Murgante, A. Lagan, D. Taniar, Y. Mun, and M. Gavrilova, Eds. Springer-Verlag, Lecture Notes in Computer Science.